Luis Joyanes Aguilar

FUNDAMENTOS DE PROGRAMACIÓN

ALGORITMOS, ESTRUCTURA DE DATOS Y OBJETOS

Quinta edición

Mc Graw Hill

FUNDAMENTOS DE

PROGRAMACIÓN

ALGORITMOS, ESTRUCTURA DE DATOS Y OBJETOS

Cristian Pilapanta 1065211-2202082223

1065211220208222

1095211220208222

FUNDAMENTOS DE

PROGRAMACIÓN

ALGORITMOS, ESTRUCTURA DE DATOS Y OBJETOS

Quinta edición

Luis Joyanes Aguilar

Dr. Ingeniero en Informática y Dr. en Sociología
Catedrático de Lenguajes y Sistemas Informáticos
Universidad Pontificia de Salamanca

Cristian Pilapanta 1095211-2202082223

10952112202082223

McGraw Hill

MÉXICO • BOGOTÁ • BUENOS AIRES • GUATEMALA • LONDRES
MADRID • MILÁN • NUEVA DELHI • NUEVA YORK • SAN JUAN
SANTIAGO • SAO PAULO • SIDNEY • SINGAPUR • TORONTO

Director general de Latinoamérica: Martín Chueco
Director editorial Latinoamérica: Hans Serrano
Gerente de portafolio de Universidad Latinoamérica: Gabriela López Ballesteros
Senior Editor: Guillermo Domínguez Chávez
Gerente de preprensa: José Palacios Hernández
Supervisor de preprensa: José Palacios

Esta publicación no puede ser reproducida ni en todo ni en parte, ni registrada en/o trasmitida por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea mecánico, fotocopiado, electrónico, ni magnético, electroóptico o cualquier otro tipo, sin el permiso previo y por escrito de la editorial.

Fundamentos de programación
Algoritmos, estructura de datos y objetos
Quinta edición

Derechos reservados copyright © 2020, 2008 respecto a la quinta edición por
McGraw-Hill Interamericana Editores, S.A. de C.V.
Prolongación Paseo de la Reforma 1015, Torre A,
Piso 16, Col. Desarrollo Santa Fe,
Alcaldía Álvaro Obregón,
CP 01376, Ciudad de México.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736.

ISBN 13: 978-1-4562-7944-8

Cristian Piñaranda 1095211-2202082226

Dedicatoria

A mis queridas nietas Inés y Olivia, quienes siempre están en mis pensamientos y cuyo gran cariño y recuerdos siempre me acompañan, tanto en mi vida profesional como en mi vida personal y familiar.

Cristian Plaparita 1065211-2202082231

1065211220208223

1095211220208222

Contenido

Prólogo a la quinta edición ..... **XIX**

PARTE I ALGORITMOS Y HERRAMIENTAS DE PROGRAMACIÓN 1

Capítulo 1 Introducción a las computadoras y a los lenguajes de programación 3

INTRODUCCIÓN 3
1.1. El origen de las computadoras y su evolución 4
1.2. Las computadoras modernas: una breve taxonomía 5
1.3. Organización de una computadora 7
1.3.1. Memoria central de la computadora 9
1.3.2. Memoria caché 10
1.3.3. Unidades de medida de memoria 11
1.3.4. Dispositivos de entrada y salida 12
1.3.5. Dispositivos de almacenamiento secundario 13
1.3.6. Sistemas de comunicación 13
1.4. Software: conceptos básicos y clasificación 13
1.4.1. Software de sistema 15
1.4.2. Software de aplicaciones 15
1.4.3. Software propietario 16
1.4.4. Software de código abierto 16
1.5. Sistema operativo 17
1.6. El lenguaje de la computadora 20
1.6.1. Representación de la información en las computadoras (códigos de caracteres) 20
1.7. La programación de las computadoras en perspectiva 21
1.8. Lenguajes de programación 22
1.8.1. Tipos de lenguajes de programación: máquina, ensamblador y de alto nivel 23
1.9. Traductores de lenguaje: el proceso de traducción de un programa 26
1.9.1. Intérpretes 26
1.9.2. Compiladores 27
1.9.3. Compiladores *versus* intérpretes 27
1.9.4. La compilación y sus fases 28
1.10. Evolución de los lenguajes de programación 29
1.11. Paradigmas de programación 30
1.12. Internet y la Web 31
1.12.1. Desarrollo de programas web 32
1.13. *Cloud computing* (computación en la nube como servicio) 33
1.13.1. Software como servicio (SaaS) 33
1.13.2. Plataforma como servicio (PaaS) 34
1.13.3. Infraestructura como servicio (IaaS) 34

Cristian Pilapanta 1095211-2202082229

1.14. Internet de las cosas 35
1.15. Big Data. Los grandes volúmenes de datos 35
1.16. Los lenguajes de programación más populares: índice TIOBE 36
1.17. Nacimiento de la programación moderna: lenguajes de programación de referencia (C, C++, Java, Python y C#) 37
RESUMEN 40

Capítulo 2 Metodología de la programación y desarrollo de software 41

INTRODUCCIÓN 41
2.1. Fases en la resolución de problemas 42
2.1.1. Análisis del problema 43
2.1.2. Diseño del algoritmo 44
2.1.3. Herramientas de programación 44
2.1.4. Codificación de un programa 47
2.1.5. Compilación y ejecución de un programa 48
2.1.6. Verificación y depuración de un programa 48
2.1.7. Documentación y mantenimiento 49
2.2. Metodología de la programación 50
2.2.1. Programación modular 50
2.3. Programación estructurada 51
2.3.1. Datos locales y datos globales 52
2.3.2. Técnicas de programación estructurada 52
2.3.3. Modelado del mundo real 53
2.4. Programación orientada a objetos 54
2.4.1. Abstracción 55
2.5. Concepto y características de algoritmos 59
2.5.1. Características de los algoritmos 60
2.5.2. Diseño del algoritmo 61
2.6. Escritura de algoritmos 63
2.7. Representación gráfica de los algoritmos 64
2.7.1. Pseudocódigo 65
2.7.2. Diagramas de flujo 66
2.7.3. Diagramas de Nassi-Schneiderman (N-S) 74
2.8. Herramientas y entornos de desarrollo de programación 76
2.8.1. Editores de texto 76
2.8.2. Programa ejecutable 76
2.8.3. Proceso de compilación/ejecución de un programa 77
2.8.4. Entorno de desarrollo integrado 78
2.8.5. Panorama de los entornos de programación 78
ACTIVIDADES DE APRENDIZAJE 79
ACTIVIDADES COMPLEMENTARIAS 79
RESUMEN 80
EJERCICIOS 80

Capítulo 3 Estructura general de un programa 83

INTRODUCCIÓN 83
3.1. Concepto de programa 84
3.2. Estructura de un programa 84
3.3. Instrucciones y tipos de instrucciones 85
3.3.1. Tipos de instrucciones 85
3.3.2. Instrucciones de asignación 86
3.3.3. Instrucciones de lectura de datos (entrada) 87
3.3.4. Instrucciones de escritura de resultados (salida) 87
3.3.5. Instrucciones de bifurcación 87
3.4. Elementos básicos de un programa 89

Cristian Pilapanta 1006211-2200892231

3.5. Datos, tipos de datos y operaciones primitivas ... 90
3.5.1. Datos numéricos ... 90
3.5.2. Datos lógicos (*booleanos*) ... 92
3.5.3. Datos tipo carácter y tipo cadena ... 92
3.6. Constantes y variables ... 92
3.6.1. Declaración de constantes y variables ... 94
3.7. Expresiones y operadores ... 94
3.7.1. Expresiones aritméticas ... 95
3.7.2. Reglas de prioridad ... 97
3.7.3. Expresiones lógicas (*booleanas*) ... 99
3.7.4. Reglas generales de prioridad y asociatividad ... 102
3.8. Funciones internas ... 102
3.8.1. Funciones matemáticas de Java ... 104
3.9. Operación de asignación ... 105
3.9.1. Asignación aritmética ... 105
3.9.2. Asignación lógica ... 106
3.9.3. Asignación de cadenas de caracteres ... 106
3.9.4. Asignación múltiple ... 106
3.9.5. Conversión de tipo ... 107
3.10. Operadores avanzados ... 108
3.10.1. Operador condicional ?: (C/C++, Java) ... 108
3.10.2. Operador coma ... 109
3.10.3. Operadores especiales: `( )`, `[ ]` ... 109
3.10.4. Operador `sizeof` ... 110
3.10.5. Operadores de manipulación de bits (*bitwise*, Java) ... 111
3.10.6. Prioridad y asociatividad ... 111
3.10.7. Conversiones de tipos ... 112
3.11. Entrada y salida de información ... 113
3.12. Escritura de algoritmos/programas ... 114
3.12.1. Cabecera del programa o algoritmo ... 114
3.12.2. Declaración de variables ... 114
3.12.3. Declaración de constantes numéricas ... 115
3.12.4. Declaración de constantes y variables carácter ... 115
3.12.5. Comentarios ... 116
3.12.6. Estilo de escritura de algoritmos/programas ... 117
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS ... 118
CONCEPTOS CLAVE ... 129
RESUMEN ... 129
EJERCICIOS ... 130

Capítulo 4 Flujo de control I: estructuras selectivas ... 133

INTRODUCCIÓN ... 133
4.1. El flujo de control de un programa ... 134
4.2. Estructura secuencial ... 134
4.3. Estructuras selectivas ... 136
4.4. Alternativa simple (`si-entonces/if-then`) ... 137
4.4.1. Alternativa doble (`si-entonces-sino/if-then-else`) ... 138
4.5. Alternativa múltiple (`según_sea, caso de; switch-case`) ... 143
4.6. Estructuras de decisión anidadas (en escalera) ... 150
4.7. La sentencia `ir-a (goto)` [no recomendable su uso, con excepciones] ... 154
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS ... 157
CONCEPTOS CLAVE ... 160
RESUMEN ... 160
EJERCICIOS ... 161

Capítulo 5 Flujo de control II: estructuras repetitivas ... 163

INTRODUCCIÓN ... 163

Cristian Pilapanta 1095211-2202092234

5.1. Estructuras repetitivas 164
5.2. Estructura mientras ("while") 166
5.2.1. Ejecución de un bucle cero veces 168
5.2.2. Bucles infinitos 169
5.2.3. Terminación de bucles con datos de entrada 169
5.3. Estructura hacer-mientras ("do-while") 171
5.4. Diferencias entre mientras (while) y hacer-mientras (do-while): una aplicación en C++ 173
5.5. Estructura repetir ("repeat") 174
5.6. Estructura desde/para ("for") 177
5.6.1. Otras representaciones de estructuras repetitivas desde/para (for) 177
5.6.2. Realización de una estructura desde con estructura mientras 180
5.7. Salidas internas de los bucles 181
5.8. Sentencias de salto interrumpir (break) y continuar (continue) 182
5.8.1. Sentencia interrumpir (break) 182
5.8.2. Sentencia continuar (continue) 183
5.9. Comparación de bucles while, for y do-while: una aplicación en C++ 184
5.10. Diseño de bucles (lazos) 185
5.10.1. Bucles para diseño de sumas y productos 185
5.10.2. Fin de un bucle 186
5.11. Estructuras repetitivas anidadas 187
5.11.1. Bucles (lazos) anidados: una aplicación en C++ 189
5.12. Sentencias de salto en C++, Java y Python: break, continue, return, goto 192
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS 194
CONCEPTOS CLAVE 206
RESUMEN 206
EJERCICIOS 207
REFERENCIAS BIBLIOGRÁFICAS 208

Capítulo 6 Subprogramas (subalgoritmos): funciones 209
INTRODUCCIÓN 209
6.1. Introducción a los subalgoritmos o subprogramas 210
6.2. Funciones 211
6.2.1. Declaración de funciones 212
6.2.2. Invocación a las funciones 213
6.3. Procedimientos (subrutinas) 218
6.3.1. Sustitución de argumentos/parámetros 219
6.4. Ámbito: variables locales y globales 222
6.5 Comunicación con subprogramas: paso de parámetros 226
6.5.1. Paso de parámetros 227
6.5.2. Paso por valor 227
6.5.3. Paso por referencia 228
6.5.4. Comparaciones de los métodos de paso de parámetros 229
6.5.5. Síntesis de la transmisión de parámetros 231
6.6. Funciones y procedimientos como parámetros 233
6.7. Los efectos laterales 235
6.7.1. En procedimientos 235
6.7.2. En funciones 236
6.8 Recursión (recursividad) 237
6.9. Funciones en C/C++, Java y C# 239
6.10. Ámbito (alcance) y almacenamiento en C/C++ y Java 241
6.11. Sobrecarga de funciones en C++ y Java 243
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS 246
CONCEPTOS CLAVE 250
RESUMEN 250
EJERCICIOS 251

Cristian Pilapanta 1095211-2202082236

PARTE II ESTRUCTURA DE DATOS 253

Capítulo 7 Estructuras de datos I (*arrays* –arreglos– y estructuras) 255
INTRODUCCIÓN 255
7.1. Introducción a las estructuras de datos 256
7.2. Arrays (arreglos) unidimensionales: los vectores 257
7.3. Operaciones con vectores 259
7.3.1. Asignación 260
7.3.2. Lectura/escritura de datos 260
7.3.3. Acceso secuencial al vector (recorrido) 261
7.3.4. Actualización de un vector 262
7.4. Arrays (arreglos) de varias dimensiones 265
7.4.1. Arrays bidimensionales (tablas/matrices) 265
7.5. Arrays (arreglos) multidimensionales 267
7.6. Almacenamiento de arrays en memoria 269
7.6.1. Almacenamiento de un vector 269
7.6.2. Almacenamiento de arrays multidimensionales 270
7.7. Estructuras *versus* registros 272
7.7.1. Registros 272
7.8. Arrays (arreglos) de estructuras 273
7.9. Uniones 275
7.9.1. Unión *versus* estructura 275
7.10. Enumeraciones 277
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS 279
CONCEPTOS CLAVE 290
RESUMEN 290
EJERCICIOS 291

Capítulo 8 Las cadenas de caracteres 293
INTRODUCCIÓN 293
8.1. Introducción 294
8.2. El juego de caracteres 294
8.2.1. Código ASCII 294
8.2.2. Código EBCDIC 295
8.2.3. Código universal Unicode para Internet 295
8.2.4. Secuencias de escape 297
8.3. Cadena de caracteres 297
8.4. Datos tipo carácter 299
8.4.1. Constantes 299
8.4.2. Variables 299
8.4.3. Instrucciones básicas con cadenas 300
8.5. Operaciones con cadenas 301
8.5.1. Cálculo de la longitud de una cadena 301
8.5.2. Comparación 302
8.5.3. Concatenación 303
8.5.4. Subcadenas 304
8.5.5. Búsqueda 305
8.6. Otras funciones de cadenas 305
8.6.1. Insertar 306
8.6.2. Borrar 306
8.6.3. Cambiar 307
8.6.4. Conversión de cadenas/números 307
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS 308
CONCEPTOS CLAVE 313

Cristian Pliqueria 1096211-2202082241

RESUMEN ... 313
EJERCICIOS ... 314

Capítulo 9 Archivos (ficheros) ... 317
INTRODUCCIÓN ... 317
9.1. Archivos y flujos (*stream*): la jerarquía de datos ... 318
9.1.1. Campos ... 319
9.1.2. Registros ... 319
9.1.3. Archivos (ficheros) ... 320
9.1.4. Bases de datos ... 320
9.1.5. Estructura jerárquica ... 320
9.1.6. Jerarquía de datos ... 320
9.2. Conceptos y definiciones: terminología ... 321
9.2.1. Clave (indicativo) ... 321
9.2.2. Registro físico o bloque ... 322
9.2.3. Factor de bloqueo ... 322
9.3. Soportes secuenciales y direccionables ... 323
9.4. Organización de archivos ... 323
9.4.1. Organización secuencial ... 324
9.4.2. Organización directa ... 324
9.4.3. Organización secuencial indexada ... 326
9.5. Operaciones sobre archivos ... 327
9.5.1. Creación de un archivo ... 328
9.5.2. Consulta de un archivo ... 328
9.5.3. Actualización de un archivo ... 329
9.5.4. Clasificación de un archivo ... 329
9.5.5. Reorganización de un archivo ... 330
9.5.6. Destrucción de un archivo ... 330
9.5.7. Reunión, fusión de un archivo ... 330
9.5.8. Rotura/estallido de un archivo ... 331
9.6. Gestión de archivos ... 331
9.6.1. Crear un archivo ... 332
9.6.2. Abrir un archivo ... 332
9.6.3. Cerrar archivos ... 334
9.6.4. Borrar archivos ... 334
9.7. Flujos ... 334
9.7.1. Tipos de flujos ... 334
9.7.2. Flujos en C++ ... 335
9.7.3. Flujos en Java ... 335
9.7.4. Consideraciones prácticas en Java y C# ... 336
9.8. Mantenimiento de archivos ... 336
9.8.1. Operaciones sobre registros ... 337
9.9. Procesamiento de archivos secuenciales (algoritmos) ... 338
9.9.1. Creación ... 338
9.9.2. Consulta ... 339
9.9.3. Actualización ... 341
9.10. Procesamiento de archivos directos (algoritmos) ... 344
9.10.1. Operaciones con archivos directos ... 344
9.10.2. Clave-dirección ... 351
9.10.3. Tratamiento de las colisiones ... 351
9.10.4. Acceso a los archivos directos mediante indexación ... 351
9.11. Procesamiento de archivos secuenciales indexados ... 353
9.12. Tipos de archivos: consideraciones prácticas en C/C++ y Java ... 353
9.12.1. Archivos de texto ... 354
9.12.2. Archivos binarios ... 355
9.12.3. Lectura y escritura de archivos ... 355

Cristian Palapanta 1095211-2202082242

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS 356
CONCEPTOS CLAVE 363
RESUMEN 363
EJERCICIOS 364

Capítulo 10 Ordenación, búsqueda e intercalación 365

INTRODUCCIÓN 365
10.1. Introducción 366
10.2. Ordenación 367
10.2.1. Método de intercambio o de burbuja 368
10.2.2. Ordenación por inserción 373
10.2.3. Ordenación por selección 375
10.2.4. Método de Shell 378
10.2.5. Método de ordenación rápida (*quicksort*) 380
10.3. Búsqueda 384
10.3.1. Búsqueda secuencial 384
10.3.2. Búsqueda binaria 389
10.3.3. Búsqueda mediante transformación de claves (*hashing*) 393
10.3.3.1. *Métodos de transformación de claves* 395
10.3.3.2. Colisiones 397
10.4. Intercalación 398
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS 401
CONCEPTOS CLAVE 413
RESUMEN 413
EJERCICIOS 414

Capítulo 11 Ordenación, búsqueda y fusión externa (archivos) 415

INTRODUCCIÓN 415
11.1. Introducción 416
11.2. Archivos ordenados 416
11.3. Fusión de archivos 416
11.4. Partición de archivos 420
11.4.1. Clasificación interna 420
11.4.2. Partición por contenido 420
11.4.3. Selección por sustitución 421
11.4.4. Partición por secuencias 423
11.5. Clasificación de archivos 424
11.5.1. Clasificación por mezcla directa 425
11.5.2. Clasificación por mezcla natural 428
11.5.3. Clasificación por mezcla de secuencias equilibradas 432
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS 433
CONCEPTOS CLAVE 437
RESUMEN 437
EJERCICIOS 438

Capítulo 12 Estructuras dinámicas lineales de datos (pilas, colas y listas enlazadas) 439

INTRODUCCIÓN 439
12.1. Introducción a las estructuras de datos 440
12.1.1. Estructuras dinámicas de datos 440
12.2. Listas 441
12.3. Listas enlazadas 443
12.4. Procesamiento de listas enlazadas 446
12.4.1. Implementación de listas enlazadas con punteros 446
12.4.2. Implementación de listas enlazadas con arrays (arreglos) 453

Cristian Palpanta 1095211-2202082242

12.5. Listas circulares ... 460
12.6. Listas doblemente enlazadas ... 461
12.6.1. Inserción ... 462
12.6.2. Eliminación ... 463
12.7. Pilas ... 463
12.7.1. Aplicaciones de las pilas ... 469
12.8. Colas ... 471
12.8.1. Representación de las colas ... 472
12.8.2. Aprovechamiento de la memoria ... 478
12.9. Doble cola ... 480
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS ... 480
CONCEPTOS CLAVE ... 489
RESUMEN ... 489
EJERCICIOS ... 489

Capítulo 13 Estructura de datos no lineales (árboles y grafos) ... 491

INTRODUCCIÓN ... 491
13.1. Introducción ... 492
13.2. Árboles ... 492
13.2.1. Terminología y representación de un árbol general ... 493
13.3. Árbol binario ... 494
13.3.1. Terminología de los árboles binarios ... 495
13.3.2. Árboles binarios completos ... 496
13.3.3. Conversión de un árbol general en árbol binario ... 497
13.3.4. Representación de los árboles binarios ... 501
13.3.4.1. Representación por punteros ... 501
13.3.4.2. Representación por listas enlazadas ... 502
13.3.4.3. Representación por arrays ... 503
13.3.5. Recorrido de un árbol binario ... 505
13.4. Árbol binario de búsqueda ... 507
13.4.1. Búsqueda de un elemento ... 509
13.4.2. Insertar un elemento ... 510
13.4.3. Eliminación de un elemento ... 511
13.5. Grafos ... 518
13.5.1. Terminología de grafos ... 519
13.5.2. Representación de grafos ... 521
13.5.2.1. Matriz de adyacencia ... 521
13.5.2.2. Lista de adyacencia ... 523
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS ... 524
CONCEPTOS CLAVE ... 529
RESUMEN ... 529
EJERCICIOS ... 530

Capítulo 14 Recursividad ... 533

INTRODUCCIÓN ... 533
14.1. La naturaleza de la recursividad ... 534
14.2. Recursividad directa e indirecta ... 538
14.2.1. Recursividad indirecta ... 541
14.2.2. Condición de terminación de la recursión ... 542
14.3. Recursión *versus* iteración ... 542
14.4. Recursión infinita ... 544
14.5. Resolución de problemas complejos con recursividad ... 549
14.5.1. Torres de Hanoi ... 549
14.5.2. Búsqueda binaria recursiva ... 555
14.5.3. Ordenación rápida (*quicksort*) ... 556
14.5.3.1. Algoritmo *quicksort* ... 560

Cristian Pilapanta 1006211-2200882243

14.5.4. Ordenación *mergesort* 560
14.5.4.1. Algoritmo *mergesort* en Java 561
CONCEPTOS CLAVE 562
RESUMEN 562
EJERCICIOS 563
PROBLEMAS 564

PARTE III PROGRAMACIÓN ORIENTADA A OBJETOS Y UML 2.5.1 565

Capítulo 15 Tipos abstractos de datos, objetos y modelado con UML 2.5.1 567
INTRODUCCIÓN 567
15.1. Programación estructurada (*procedimental*) 568
15.1.1. Limitaciones de la programación estructurada 568
15.1.2. Modelado de objetos del mundo real: el camino a los objetos 569
15.2. Programación orientada a objetos 570
15.2.1. Objetos 571
15.2.2. Tipos abstractos de datos: CLASES 572
15.3. Modelado e identificación de objetos 574
15.4. Propiedades fundamentales de orientación a objetos 575
15.4.1. Abstracción 575
15.4.2. La abstracción en el *software* 576
15.4.3. Encapsulamiento y ocultación de datos 576
15.4.4. Herencia 577
15.4.5. Reutilización o *reusabilidad* 578
15.4.6. Polimorfismo 578
15.5. Modelado de aplicaciones: UML 579
15.5.1. Lenguaje de modelado 580
15.5.2. UML: el Lenguaje Unificado de Modelado 581
15.6. Modelado y modelos 581
15.7. Diagramas de UML 2.5.1 583
15.8. Bloques de construcción (componentes) de UML 2.5.1 586
15.8.1. Elementos estructurales 586
15.8.2. Elementos de comportamiento 587
15.8.3. Elementos de agrupación 588
15.9. Especificaciones de UML 588
15.10. Historia de UML 589
CONCEPTOS CLAVE 590
RESUMEN 590
ANEXO: TERMINOLOGÍA DE ORIENTACIÓN A OBJETOS 591
EJERCICIOS 591

Capítulo 16 Diseño de clases y objetos: representaciones gráficas en UML 593
INTRODUCCIÓN 593
16.1. Diseño y representación gráfica de objetos en UML 594
16.1.1. Representación gráfica en UML 595
16.1.2. Características de los objetos 596
16.1.3. Estado 597
16.1.4. Múltiples instancias de un objeto 599
16.1.5. Evolución de un objeto 599
16.1.6. Comportamiento 600
16.1.7. Identidad 602
16.1.8. Los mensajes 602
16.1.9. Responsabilidad y restricciones 604
16.2. Diseño y representación gráfica de clases en UML 604
16.2.1. Representación gráfica de una clase 605

Cristian Pilapanta 1095211-2202082245

16.2.2. Declaración de una clase 608
16.2.3. Reglas de visibilidad 610
16.2.4. Sintaxis 611
16.3. Declaración de objetos de clases 612
16.3.1. Acceso a miembros de la clase: encapsulamiento 614
16.3.2. Declaración de métodos 616
16.3.3. Tipos de métodos 619
16.4. Constructores 619
16.4.1. Constructor por defecto (predeterminado) 620
16.5. Destructores 623
16.6. Implementación de clases en C++ 625
16.6.1. Archivos de cabecera y de clases 625
16.6.2. Clases compuestas 626
16.7. Recolección de basura 627
16.7.1. El método `finalize()` 628
CONCEPTOS CLAVE 628
RESUMEN 629
EJERCICIOS 630
LECTURAS RECOMENDADAS 632

Capítulo 17 Relaciones entre clases: delegaciones, asociaciones, agregaciones, herencia 633
INTRODUCCIÓN 633
17.1. Relaciones entre clases 634
17.2. Dependencia 635
17.3. Asociación 635
17.3.1. Multiplicidad 637
17.3.2. Restricciones en asociaciones 638
17.3.3. Asociación cualificada 639
17.3.4. Asociaciones reflexivas 639
17.3.5. Diagrama de objetos 639
17.3.6. Clases de asociación 640
17.3.7. Restricciones en asociaciones 644
17.4. Agregación 645
17.4.1. Composición 647
17.5. Jerarquía de clases: generalización y especialización 648
17.5.1. Jerarquías de generalización/especialización 650
17.6. Herencia: clases derivadas 653
17.6.1. Herencia simple 654
17.6.2. Herencia múltiple 654
17.6.3. Niveles de herencia 655
17.6.4. Declaración de una clase derivada 658
17.6.5. Consideraciones de diseño 658
17.7. Accesibilidad y visibilidad en herencia 659
17.7.1. Herencia pública 659
17.7.2. Herencia privada 660
17.7.3. Herencia protegida 660
17.8. Un caso de estudio especial: herencia múltiple 661
17.8.1. Características de la herencia múltiple 663
17.9. Clases abstractas 665
17.9.1. Operaciones abstractas 666
CONCEPTOS CLAVE 667
RESUMEN 667
EJERCICIOS 667

Cristian Pilapanta 1096211-2200882246

PARTE IV METODOLOGÍA DE LA PROGRAMACIÓN Y DESARROLLO DE SOFTWARE 671

Capítulo 18 Ingeniería de software y metodología de la programación 673
INTRODUCCIÓN 673
18.1. Introducción a la ingeniería de software: técnicas y modelos 674
18.1.1. Software propietario 675
18.1.2. Software abierto: código abierto (*open source*) 675
18.2. Proceso de desarrollo de software 676
18.2.1. Actividades del proceso de desarrollo de software 677
18.3. Ciclo de vida del desarrollo de software 678
18.4. Modelos de procesos de desarrollo de software 681
18.4.1. Métodos ágiles 682
18.4.2. Prototipado 683
18.5. Ciclo de vida del software "en cascada" (*waterfall*) 683
18.5.1. Modelo *waterfall* de Sommerville 683
18.5.2. Modelo de cascada *waterfall* de Pressman 684
18.5.3. Modelo tradicional de ciclo de vida del software "en cascada" 685
18.6. Metodología de la programación: análisis de especificaciones y requerimientos 686
18.7. Diseño 688
18.7.1. Diseño de algoritmos 690
18.8. Implementación (codificación) 691
18.9. Pruebas e integración 691
18.9.1 Verificación 691
18.9.2. Técnicas de pruebas 692
18.9.3. Revisión general de validación y verificación de programas 694
18.10. Depuración de programas 696
18.10.1. Localización y reparación de errores 696
18.10.2. Depuración de sentencias `si-entonces-sino` 697
18.10.3. Los equipos de programación 698
18.10.4. El depurador 698
18.11. Mantenimiento 698
18.11.1. La obsolescencia: programas obsoletos 699
18.11.2. Iteración y evolución del software 699
18.12. Documentación 700
18.12.1. Manual del usuario 700
18.12.2. Manual de mantenimiento (documentación para programadores) 701
18.12.3. Reglas de documentación 702
18.13. Depuración 702
18.13.1. Localización y reparación de errores 703
18.13.2. Depuración de sentencias `si-entonces-si no` 703
18.13.3. Los equipos de programación 704
18.14. Diseño de algoritmos 704
18.15. Eficiencia 705
18.15.1. Eficiencia *versus* legibilidad (claridad) 705
18.16. Portabilidad 706
CONCEPTOS CLAVE 706
RESUMEN 706

Bibliografía y recursos de programación 709

Cristian Pilapanta 1005211-2202082247

1095211220208224

Prólogo a la quinta edición

Desde 2008, año de la publicación de la cuarta edición de esta obra, mucho ha cambiado el mundo en general y el mundo de la programación en particular. Numerosas tecnologías de la información y las comunicaciones (TIC) se han consolidado y, sobre todo, numerosas tecnologías disruptivas han aparecido. De igual forma se han consolidado nuevas tendencias tecnológicas como el caso de la **Industria 4.0** y la **Cuarta Revolución Industrial**, que han traído consigo el asentamiento y despliegue de *innumerables* tendencias tecnológicas como *Inteligencia Artificial, Big Data, Cloud Computing, Internet de las Cosas, Blockchain, Computación Cuántica, Redes Celulares –especialmente redes 4G y la naciente 5G–, Ciberseguridad...* unidas al despegue creciente de un nuevo **mundo ubicuo**, donde *todas las cosas serán inteligentes, se podrán conectar entre sí, y se podrán conectar a Internet en cualquier lugar, en cualquier momento y con cualquier dispositivo.*

Todas las ediciones de esta obra, y naturalmente esta quinta edición, se han apoyado en las directrices recomendadas por las organizaciones nacionales e internacionales de España, Latinoamérica, el Caribe, y Portugal y Brasil –países donde prolifera el estudio del lenguaje español y donde algunos de nuestros libros han sido traducidos al idioma portugués.

Cristian Pilapanta 1096211-2202082249

Computing Curricula

Como norma general, siempre se han tratado de seguir también las directrices de las organizaciones internacionales de computación (*computer science*) y en particular, la **ACM** (The Association for Computing Machinery), y **la IEEE-CS** (The Computer Society). Las recomendaciones para el estudio de carreras relacionadas con *Computer Science*, han constituido y constituyen la referencia obligada en casi todos los países de lengua española y, naturalmente, de lengua inglesa. Las cuatro primeras ediciones trataron de adaptarse a las directrices curriculares de la ACM/IEEE conocidas como *Computing Curricula*, tomando como base las últimas publicadas esencialmente, *Computing Curricula 1991* o CC1991 y *Computing Curricula 2001* o CC2001. Sin embargo, es el informe Computing *Curricula 2005*[1] publicado en 2006, el que contempla la actualización que hoy día sigue la mayoría de las universidades, institutos tecnológicos, centros de formación, ya que propone no una única disciplina de *computer science* sino cinco disciplinas o subdisciplinas y que de una u otra forma están adoptando todas las organizaciones académicas y profesionales: Ingeniería de Computadores (Computer Engineering, CE), presta la atención en los dispositivos de hardware; Ingeniería del Software (Software Engineering, SE), el énfasis está en la creación de software que satisfaga las robustas necesidades del mundo real y Ciencias de la Computación (Computer Science, CS), y con el enfoque del software como "lenguaje" con el que se expresan las ideas y se explora una amplia gama de problemas de computación y aplicaciones, las disciplinas de Tecnologías de la Información (*Information Technologies*, IT) y Sistemas de Información (*Information Systems*, IS)

[1] Computing Curricula 2005 (CC2005). The Overview Report covering undergraduate degree programs in: *Computer Engineering, Computer Science, Information Systems, Information Technology, Software Engineering*. https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2005-march06final.pdf

Las cinco carreras citadas anteriormente tienen como primera asignatura recomendada en todas ellas la asignatura Programming Fundamentals.[2] Los planes de estudios de ingeniería informática en España[3] (grado en ingeniería informática) y de otras ingenierías (telecomunicaciones, industriales, etcétera) y de ingeniería de sistemas en Latinoamérica, incluyen asignaturas tales como Fundamentos de Programación, o nombres similares como Metodología de la Programación, Introducción a la Programación o Programación de computadoras.

La ACM/IEEE-CS está trabajando en la actualidad para publicar el siguiente *Computing Curricula 2020 (CC2020)*[4] que englobará las cinco disciplinas anteriores y una sexta centrada en Ciberseguridad, aunque durante estos últimos años ha publicado la mayoría de las nuevas directrices propuestas en CC 2005:[5]

- CC2005 – *Computing Curricula*
- SE2014 – *Software Engineering*
- CS2013 – *Computer Science*
- CE2016 – *Computer Engineering*
- IT2017 – *Information Technology*
- CSEC2017 – *Cybersecurity*

Computer Society está realizando un nuevo proyecto curricular sobre *Data Science* (Ciencia de Datos) en CC2020.[6] La informática y las ciencias de la computación en los primeros años del siglo XXI vienen marcadas por los avances tecnológicos de la década pasada. Los más de veinte años de vida de la computadora personal (**PC**) y los más de cincuenta años de la informática/computación tradicional vienen acompañados de cambios rápidos y evolutivos en las disciplinas clásicas. El rápido crecimiento del mundo de las redes y, en consecuencia, la **World Wide Web** hacen revolucionarios a estos cambios y afectan al cuerpo de conocimiento de los procesos educativos y profesionales.

Del estudio comparado de las citadas recomendaciones curriculares, así como de planes de estudios conocidos de carreras de ingeniería de sistemas y licenciaturas en informática de universidades latinoamericanas, hemos llegado a la consideración de que la iniciación de un estudiante de ingeniería informática o de ingeniería de sistemas en las técnicas de programación actuales y futuros requieren no sólo del aprendizaje clásico del diseño de algoritmos y de la comprensión de las técnicas orientadas a objetos, sino de un **método de transición hacia las tecnologías predominantes en el horizonte 2020 y las tecnologías disruptivas de la Industria 4.0 y la Cuarta Revolución Industrial.** Por otra parte, un libro dirigido a los primeros cursos de introducción a la programación exige no sólo la elección de un lenguaje de programación adecuado, sino, y sobre todo, «proporcionar al lector las herramientas para desarrollar programas correctos, eficientes, bien estructurados y con estilo, que sirvan de base para la construcción de unos fundamentos teóricos y prácticos que le permitan continuar estudiando con éxito los cursos superiores de su carrera, así como una futura especialización en ciencias e ingeniería». En consecuencia y de modo global, la obra pretende enseñar técnicas de **análisis**, **diseño y construcción de algoritmos**, **estructuras de datos** y **objetos**, así como reglas para la escritura de **programas**, **eficientes tanto estructurados**, **fundamentalmente**, **como orientados a objetos**. De modo complementario, y no por ello menos importante, se busca también enseñar al alumno técnicas de abstracción que le permitan resolver los problemas de programación del modo más sencillo y racional pensando no sólo en el aprendizaje de reglas de sintaxis y construcción de programas, sino, y sobre todo, aprender a pensar para conseguir la resolución del problema en cuestión de forma clara, eficaz y fácil de implementar en un lenguaje de programación y su ejecución posterior en una computadora u ordenador.

Lectores a los que va dirigido este libro

El libro está dirigido, principalmente, a estudiantes de universidad, institutos tecnológicos y centros de formación profesional, y estudiantes/profesionales autodidactas, que toman cursos introductores y avanzados en ingeniería de sistemas, ingeniería y licenciatura en informática, y restantes ingenierías y carreras de ciencias, otros grados en

[2] Table 3.1: Comparative weight of computing topics across the five kinds of degree programs.
[3] El profesor de la Universidad de Salamanca F. J. García-Peñalvo es autor de un informe muy completo y de gran calidad sobre: *Los estudios de Ingeniería en Informática* un *Proyecto Docente e Investigador.* Salamanca, España: Departamento de Informática y Automática. Universidad de Salamanca, 2018. https://repositorio.grial.eu › bitstream › grial › ProyectoDocenteInvestigador.
[4] https://www.computer.org/volunteering/boards-and-committees/professional-educational-activities/curricula.
[5] https://www.acm.org/education/curricula-recommendations
[6] CC2020-net

Cristian Pliqueria 1096211-2202082250

formación profesional, en los que se cursan asignaturas tales como Fundamentos de Programación, Introducción a la Programación, Metodología de programación y otros términos o acepciones similares como Programación I y II, Algoritmos y Estructuras de Datos, entre otras.

El contenido ha sido actualizado pero adaptado a los objetivos de las ediciones anteriores: llegar a estudiantes de programación de computadoras en sus niveles de iniciación y medio, o a nivel superior que deseen consolidar sus conocimientos con los lenguajes de programación tradicionales C, C++, Java o se están sumergiendo en los nuevos lenguajes de programación que hoy están llegando a los departamentos de desarrollo de software en organizaciones y empresas, o en departamentos universitarios y tecnológicos, como Python, R o JavaScript y que también necesitan afrontar nuevos lenguajes como Swift, Go o Scala.

Objetivos del libro

El libro pretende enseñar a programar utilizando conceptos fundamentales, como:

1. *Algoritmos* (conjunto de instrucciones programadas para resolver una tarea específica).
2. *Datos* (colección de datos que se proporciona a los algoritmos que se deben ejecutar para encontrar una solución: los datos se organizarán en *estructuras de datos*).
3. *Objetos* (conjunto de datos y algoritmos que los manipulan, encapsulados en un tipo de dato conocido como **objeto**).
4. *Clases* (tipos de objetos con igual estado y comportamiento o, dicho de otro modo, los mismos atributos y operaciones).
5. *Estructuras de datos* (conjunto de organizaciones de datos para tratar y manipular eficazmente datos homogéneos y heterogéneos).
6. *Temas avanzados* (recursividad, métodos avanzados de ordenación y búsqueda, relaciones entre clases, entre otros).
7. *Programación Orientada a Objetos* (propiedades fundamentales, diagramas UML 2.5.1, diseño de clases, etcétera).
8. *Desarrollo de proyectos de software* (introducción al ciclo de vida del software, ingeniería de software y guías prácticas de metodología de la programación).

Los dos primeros aspectos, algoritmos y datos, han permanecido invariables a lo largo de la corta historia de la informática/computación, pero la *interrelación* entre ellos sí que ha variado y continuará haciéndolo. Esta interrelación se conoce como *paradigma de programación*. Sin embargo, el binomio algoritmos-datos ha alcanzado cimas impensables hace unos años, tales como su aplicación en las técnicas y métodos de Big Data, y en la aplicación de los algoritmos en las técnicas de aprendizaje automático (*machine learning*) y aprendizaje profundo (*deep learning*) dentro de la ciencia de la Inteligencia Artificial.

En el paradigma de programación *procedimental* (*procedural* o *por procedimientos*) un problema se modela directamente mediante un conjunto de algoritmos. Por ejemplo, la nómina de una empresa o la gestión de ventas de un almacén se representan como una serie de funciones que manipulan datos; mismos que se almacenan separadamente y se accede a ellos, o bien mediante una posición global o mediante parámetros en los procedimientos. Tres lenguajes de programación clásicos, Fortran, Pascal y C, han representado el arquetipo de la programación *procedimental*, también relacionada estrechamente y —normalmente— conocida como **programación estructurada**. La programación con soporte en C y Pascal proporciona el paradigma *procedimental* tradicional con un énfasis en funciones, plantillas de funciones y algoritmos genéricos.

En la década de los ochenta, el enfoque del diseño de programas se desplazó desde el paradigma *procedimental* al *orientado a objetos* apoyado en los tipos abstractos de datos (TAD); desde entonces conviven los dos paradigmas. En el paradigma orientado a objetos un problema se modela en un conjunto de abstracciones de datos (tipos de datos) conocidos como *clases*. Las clases contienen un conjunto de instancias o ejemplares de la misma que se denominan *objetos*, de modo que un programa actúa como un conjunto de objetos que se relacionan entre sí. La gran diferencia entre ambos paradigmas reside en el hecho de que los algoritmos asociados con cada clase se conocen como *interfaz pública* de la clase y los datos se almacenan privadamente dentro de cada objeto, de modo que el acceso a los datos está oculto al programa general y se gestionan a través de la interfaz.

Así pues, en resumen, los objetivos fundamentales de esta obra son: aprendizaje y formación en *algoritmos y programación estructurada, estructuras de datos y programación orientada a objetos*. Evidentemente, la mejor forma de aprender a programar es con la ayuda de un lenguaje de programación. Apoyados en la experiencia de nuestras

Cristian Palpanta 1095211-2202082250

cuatro primeras ediciones y en los resultados conseguidos con nuestros alumnos y lectores, hemos seguido apostando por utilizar un lenguaje algorítmico —**pseudolenguaje**— que apoyado en un *pseudocódigo (seudocódigo)* en español nos permitiera enseñar al alumno las técnicas y reglas de programación y que su aprendizaje fuese rápido y gradual. Naturalmente, además del **pseudocódigo** hemos utilizado las otras herramientas de programación clásicas y probadas como los **diagramas de flujo** o los **diagramas N-S**.

En esta quinta edición hemos seguido utilizando el mismo lenguaje algorítmico al que ya se le añadió las construcciones y estructuras necesarias para incorporar en sus especificaciones las técnicas orientadas a objetos. Así, hemos seguido utilizando nuestro lenguaje algorítmico UPSAM inspirado en su origen en los lenguajes estructurados por excelencia, y a los que añadimos en su momento propiedades de los lenguajes orientados a objetos tales como C++, Python, Java y C#, en la mejor armonía posible y con unas especificaciones que hemos incluido en los apéndices de la edición en papel y en su sitio web.

El libro como herramienta docente

En el contenido de la obra hemos tenido en cuenta no sólo las directrices de los planes de estudio españoles de los grados en ingeniería informática y licenciaturas en ciencias, sino también de ingenierías, tales como industriales, telecomunicaciones, agrónomos o minas, o las más jóvenes, como ingeniería en geodesia, ingeniería química o ingeniería telemática. Nuestro conocimiento del mundo educativo latinoamericano nos ha llevado a pensar también en las carreras de ingeniería de sistemas computacionales y las licenciaturas en informática y en sistemas de información, como se las conoce en Latinoamérica.

El contenido del libro se ha escrito pensando en un posible desarrollo de dos cuatrimestres o semestres o en un año completo, y siguiendo los descriptores (temas centrales) recomendados en las directrices del Ministerio de Educación y Ciencia español y ministerios latinoamericanos homónimos (México, Colombia, Perú, Argentina, Chile, República Dominicana, Panamá, Guatemala, etcétera) para los planes de estudio de grado en **Informática** y grados en **Ingeniería Industrial**, de **Telecomunicaciones**, y los planes de estudios de **Ingeniería de sistemas** y licenciaturas en **Informática** con el objeto de poder ser utilizado también por los estudiantes de primeros semestres de estas carreras. En asignaturas como Fundamentos de Programación, Metodología de la Programación e Introducción a la Programación, y también, si el lector o el maestro/profesor lo consideran oportuno, para cursos de Introducción a Estructuras de Datos y/o a Programación Orientada a Objetos.

El contenido del libro trata de abarcar los contenidos curriculares de las asignaturas citadas y comienza con la introducción a los algoritmos y a la programación, para llegar a estructuras de datos y programación orientada a objetos. Por esta circunstancia la estructura del curso no ha de ser secuencial en su totalidad, sino que el profesor/maestro y el alumno/lector podrán estudiar sus materias en el orden que consideren más oportuno. Ésta es la razón principal por la cual el libro se ha organizado en cuatro partes y en un gran número de apéndices que complementen los contenidos del libro y proporcionen el mayor número posible de recursos de programación tanto a estudiantes como profesionales.

Se trata de describir los *dos paradigmas* más populares en el mundo de la programación: el *procedimental* y el *orientado a objetos*. Los cursos de programación en sus niveles inicial y medio están evolucionando para aprovechar las ventajas de nuevas y futuras tendencias en ingeniería de software y en diseño de lenguajes de programación, específicamente diseño y programación orientada a objetos. Numerosas facultades y escuelas de ingenieros, junto con la **nueva formación profesional** (*ciclos formativos de nivel superior*) en España y en Latinoamérica, están introduciendo a sus alumnos en la programación orientada a objetos, inmediatamente después del conocimiento de la programación estructurada, e incluso —en ocasiones antes—. Por esta razón, una metodología que se podría seguir sería impartir un curso de *algoritmos e introducción a la programación* (parte I) seguido de *estructuras de datos* (parte II) y luego seguir con un segundo nivel de *programación avanzada y programación orientada a objetos* (partes II, III y IV) que constituyen las cuatro partes del libro. De modo complementario, si el alumno o el profesor/maestro lo desea se puede practicar con algún lenguaje de programación estructurado u orientado a objetos, y pensando en esa posibilidad se incluyen en la página web del libro, apéndices con guías de sintaxis de los lenguajes de programación más populares hoy como Java, C, C++, Python y C# y referencias a otros lenguajes de programación cada vez más utilizados como R, Ruby, Scala o Swift.

Uno de los temas más debatidos en la educación en informática o en ciencias de la computación (*Computer Sciences*) es el rol de la programación en el currículo introductorio. A través de la historia de la disciplina, la mayoría de los cursos de introducción a la informática se han centrado principalmente en el desarrollo de habilidades o

Cristian Pilapanta 1096211-2202082251

destrezas de programación. La adopción de un curso de nivel básico y medio de programación proviene de una serie de factores prácticos e históricos e incluyen los siguientes temas:

- La programación es una técnica esencial que debe ser dominada por cualquier estudiante de informática, de computación y de ingeniería de sistemas. Su inserción en los primeros cursos de la carrera asegura que los estudiantes tengan la facilidad necesaria con la programación para cuando se matriculan en los cursos de nivel intermedio y avanzado.
- La informática no se convirtió en una disciplina académica hasta después de que la mayoría de las instituciones ha desarrollado un conjunto de cursos de programación introductoria que sirva a una gran audiencia.
- El modelo de aprendizaje en programación ha seguido las directrices emanadas de los *computing curricula* de la ACM/IEE-CS.
- Los programas de informática deben enseñar a los estudiantes cómo utilizar de manera adecuada al menos un lenguaje de programación. Además, recomendamos que los programas en informática enseñen a los estudiantes a ser competentes en lenguajes y que hagan uso de, por lo menos, dos paradigmas de programación. Como consecuencia de estas ideas, se contempla la necesidad de conceptos y habilidades que son fundamentales en la práctica de la programación con independencia del paradigma subyacente. Como resultado de este pensamiento, el área de Fundamentos de Programación incluye unidades sobre conceptos de programación, estructuras de datos básicas y procesos algorítmicos. Además de estas unidades fundamentales se requieren conceptos básicos pertenecientes a otras áreas, como son Lenguajes de Programación (PL, Programming Languages) y de Ingeniería de Software (SE, Software Engineering).

Los temas fundamentales que se consideraban en el currículo 2001 y han seguido en el de 2005 de la ACM, y seguirán en el 2020, son:

- Construcciones fundamentales de programación.
- Algoritmos y resolución de problemas.
- Estructuras de datos fundamentales.
- Recursión o recursividad.
- Programación controlada por eventos.

De otras áreas, Lenguajes de Programación (PL), destacan:

- Revisión de lenguajes de programación.
- Declaraciones y tipos.
- Mecanismos de abstracción.
- Programación orientada a objetos.

Y de Ingeniería de Software (SE):

- Diseño de software.
- Herramientas y entornos de software.
- Requisitos y especificaciones de software.
- Desarrollo de procesos de software.

Este libro se ha escrito pensando en que pudiera servir de referencia y guía de estudio para un primer curso de Introducción a la programación, con una segunda parte que, a su vez, sirviera como continuación y para un posible segundo curso de *Estructuras de datos* y *programación orientada a objetos*. El objetivo final que busca es, no sólo describir la sintaxis de un lenguaje de programación, sino, y sobre todo, mostrar las características más sobresalientes del lenguaje algorítmico y a la vez enseñar técnicas de programación estructurada y orientada a objetos. Así pues, los objetivos fundamentales son:

- Énfasis fuerte en el análisis, construcción y diseño de programas.
- Un medio de resolución de problemas mediante técnicas de programación.
- Una introducción a la informática y a las ciencias de la computación usando una herramienta de programación denominada **pseudocódigo**.
- Aprendizaje de técnicas de construcción de programas estructurados y una iniciación a los programas orientados a objetos.

Cristian Pliqueria 1095211-2202082257

Así, se tratará de enseñar las técnicas clásicas y avanzadas de programación estructurada, junto con técnicas orientadas a objetos. La programación orientada a objetos es la panacea universal de un programador del siglo XXI y un paradigma imprescindible para los programadores y desarrolladores de software que demandan las empresas.

El contenido del libro trata de proporcionar soporte a un año académico completo (dos semestres o cuatrimestres), alrededor de 24 a 32 semanas, dependiendo lógicamente de su calendario y planificación. Los diez primeros capítulos pueden comprender el primer semestre y los capítulos restantes pueden impartirse en el segundo semestre. Lógicamente la secuencia y planificación real dependerá del profesor que marcará y señalará semana a semana la progresión que él considere lógica. Si usted es un estudiante autodidacto, su propia progresión vendrá marcada por las horas que dedique al estudio y al aprendizaje con la computadora, aunque no debe variar mucho del ritmo citado al principio de este párrafo.

Cambios de la quinta edición

Un resumen de las actualizaciones y adiciones de contenidos de esta quinta edición son:

- Actualización de las nuevas propiedades básicas incorporadas en las versiones de lenguajes de programación más utilizados en la actualidad.
- La Parte I ha sufrido una gran reestructuración para adaptarla a las características principales de las computadoras utilizadas en organizaciones y empresas, y que soportarán las prácticas de programación.
- La Parte II se ha revisado en profundidad, para detectar erratas o actualizar contenidos que puedan ser de interés en la formación docente y, en su caso, añadir las nuevas características fundamentales que incorporan los lenguajes de programación de aprendizaje.
- La Parte III "Programación Orientada a Objetos y UML 2.5.1" se ha reestructurado añadiendo las nuevas propiedades y características incorporadas a las últimas versiones de UML normalizadas, como la OMG 2.5.1 (última versión actualizada por OMG a finales de septiembre de 2019).
- La Parte IV se ha reestructurado, también, en su totalidad, dados los avances en el desarrollo de proyectos de software obligatoriamente introducidos por organizaciones y empresas. Para ello se han introducido secciones específicas de introducción a la ingeniería de software, ciclo de vida del software y metodologías o modelos de proyectos de desarrollo de software.
- Los apéndices se han actualizado para esta edición y se encuentran en el sitio web del libro: `www.mhhe.com/latam/joyanes_fp5e`.
- Se ha actualizado la bibliografía y recursos web utilizados en la nueva redacción de contenidos, y en bibliografía de lenguajes de programación, fundamentalmente, Python, además claro de novedades en C/C++, Java y UML.

Los contenidos del sitio web se han organizado para ofrecer al lector (alumnos, profesores y profesionales) documentación de interés para completar sus estudios en programación, así como guías de aprendizaje, herramientas, recursos de programación, cursos de aprendizaje, etcétera a diferentes niveles, de modo que pueda permitir al lector actualizarse en las exigencias y competencias que le requerirán en las empresas donde sean contratados como becarios en prácticas o ya como profesionales. Por otro lado, para los docentes y profesionales se ofrecen recursos bibliográficos y web que les permitan conocer herramientas y recursos que puedan utilizar en su actividad diaria y complementen sus contenidos. En esencia, se trata de ofrecer una gran cantidad de recursos que ayuden al lector, estudiante, egresado, profesional o académico a disponer de información exhaustiva que lo apoye en las ingentes tareas de aprendizaje de la programación y en el desarrollo de algoritmos y programas. Se han actualizado los recursos web a la nueva década y se han añadido nuevos y completos contenidos para agilizar su aprendizaje y profundización en técnicas de programación y en lenguajes de programación, especialmente C, C++, Java y Python, además de herramientas de UML y de Ingeniería de Software.

Características importantes del libro

Esta nueva edición utiliza en cada capítulo los siguientes elementos clave para obtener el mayor rendimiento del material incluido:

- **Contenido**. Índice completo del capítulo que facilita la lectura y la correcta progresión en la lectura y comprensión de los diferentes temas que se exponen posteriormente.

Cristian Platania 1006211-2200082258

- **Introducción**. Abre el capítulo con una breve revisión de los puntos y objetivos más importantes que se tratarán y todo aquello que se puede esperar del mismo.
- **Descripción del capítulo**. Explicación usual de los apartados correspondientes del capítulo. En cada capítulo se incluyen ejemplos y ejercicios resueltos. Los listados de los programas completos o parciales se escriben en letra `"courier"` con la finalidad principal de que puedan ser identificados fácilmente por el lector. Todos ellos han sido probados para facilitar la práctica del lector/alumno.
- **Conceptos clave**. Enumera los términos de computación, informáticos y de programación (terminología) más notables que se han descrito o tratado en el capítulo.
- **Actividades de programación resueltas**. Enunciados de problemas de programación incluyendo el análisis del problema y el diseño del algoritmo utilizando la herramienta del pseudocódigo normalizado a lo largo del libro y conocido como leguaje algorítmico de programación UPSAM 2.0
- **Resumen del capítulo**. Revisa los temas importantes que los estudiantes y lectores deben comprender y recordar. Busca también ayudar a reforzar los conceptos clave que se han aprendido en el capítulo.
- **Ejercicios**. Al final de cada capítulo se proporciona a los lectores una lista de ejercicios sencillos de modo que le sirvan de oportunidad para que puedan medir el avance experimentado mientras leen y siguen, en su caso, las explicaciones del profesor relativas al capítulo.
- **Problemas**. En muchos capítulos se incluyen enunciados de problemas propuestos para realizar por el alumno y que presentan una mayor dificultad que los ejercicios antes planteados. Se suele incluir una serie de actividades y proyectos de programación que se le proponen al lector como tarea complementaria de los ejercicios.

A lo largo de todo el libro se incluye una serie de recuadros —sombreados o no— que ofrecen al lector consejos, advertencias y reglas de uso del lenguaje y de técnicas de programación, con la finalidad de que puedan ir asimilando conceptos prácticos de interés que les ayuden en el aprendizaje y construcción de programas eficientes y de fácil lectura.

- **Recuadro**. Conceptos importantes que el lector debe considerar durante el desarrollo del capítulo.
- **Consejo**. Ideas, sugerencias, recomendaciones al lector, con el objetivo de obtener el mayor rendimiento posible del lenguaje y de la programación.
- **Precaución**. Advertencia al lector para que tenga cuidado al hacer uso de los conceptos incluidos en el recuadro adjunto.
- **Nota**. Normas o ideas que el lector debe seguir preferentemente en el diseño y construcción de sus programas.

Organización del libro

El libro se ha dividido en cuatro partes a efectos de organización para su lectura y estudio gradual. Dado que el conocimiento es acumulativo, se comienza en los primeros capítulos con aspectos conceptuales y prácticos, y se avanza de modo progresivo hasta llegar a las técnicas avanzadas y a una introducción a la ingeniería de software que intentan preparar al lector/estudiante para sus estudios posteriores. Como complemento y ayuda al lector durante sus estudios y para su posterior formación profesional, se ha incluido una gran cantidad de apéndices que incluyen fundamentalmente guías de sintaxis de los lenguajes de programación más populares con el objetivo de facilitar la implementación de los algoritmos en el lenguaje de programación elegido para «dialogar» con la computadora. Asimismo, y para ayudar a la preparación del aprendizaje, lecturas y estudios futuros, se ha incluido una amplia guía de recursos de programación (libros, revistas y sitios web URL) que hemos consultado en la elaboración de nuestra obra y seguimos consultando también en nuestra vida profesional.

Parte I. Algoritmos y herramientas de programación

Esta parte es un primer curso de programación para alumnos principiantes en asignaturas de introducción a la programación tanto en lenguajes estructurados como lenguajes orientados a objetos y sirve tanto para cursos introductorios de carácter semestral, como *Introducción a la programación, Metodología de la programación* o *Fundamentos de programación,* en primeros cursos de carreras de ingeniería informática, ingeniería de sistemas y licenciatura en informática o en sistemas de información, y asignaturas de programación de ingeniería y de ciencias. Contiene esta parte los fundamentos teóricos y prácticos relativos a la organización de una computadora y los lenguajes de programación, así como la descripción de las herramientas de programación más frecuentemente utilizadas en el cam-

Cristian Pilapanta 1096211-2202082259

po de la programación. Se incluyen también en esta parte los elementos básicos constitutivos de un programa y las herramientas de programación utilizadas, tales como algoritmos, diagramas de flujo, entre otras. La segunda mitad de esta primera parte es una descripción teórico-práctica de las estructuras utilizadas para controlar el flujo de instrucciones de un programa y una descripción detallada del importante concepto de subprograma (procedimiento/función), piedra angular de la programación modular y estructurada.

Capítulo 1. ***Introducción a las computadoras y a los lenguajes de programación.*** Este capítulo introductorio describe la organización de una computadora y sus diferentes partes junto con los conceptos de programa y de lenguaje de programación. Además, y con el objetivo de que el lector pueda entender los fundamentos teóricos en que se asienta la programación, se incluye una breve historia de los lenguajes de programación más influyentes y que, en el caso de la tercera edición, han servido de inspiración para la nueva versión del pseudocódigo UPSAM 2.5: es decir, Java, C, C++, Python y C#, con referencias lógicas al histórico Pascal –todavía bastante utilizado según el prestigioso índice de lenguajes de programación, Tiobe.

Capítulo 2. ***Metodología de la programación y desarrollo de software.*** En este capítulo se describen métodos para la resolución de problemas con computadora y con un lenguaje de programación (en nuestro caso el pseudolenguaje o lenguaje algorítmico UPSAM 2.5). Se explican las fases de la resolución de un problema junto con las técnicas de programación modular y estructurada. Se inicia en este capítulo la descripción del concepto, función y uso de algoritmos ya que es uno de los objetivos más importantes de este libro.

Capítulo 3. ***Estructura general de un programa.*** Enseña la organización y estructura general de un programa, así como su creación y proceso de ejecución. Se describen los elementos básicos de un programa: tipos de datos, constantes, variables y entradas/salidas de datos. También, se introduce al lector en la operación de asignación, así como en el concepto de función interna. De igual forma se estudian los conceptos de expresiones y operaciones junto con sus diferentes tipos.

Capítulo 4. ***Flujo de control I: estructuras selectivas.*** Introduce al concepto de estructura de control y, en particular, estructuras de selección, tales como `si-entonces ("if-then")`, `según_sea/caso_de ("switch/case")`. Se describen también las estructuras de decisión anidadas. Así mismo se explica también la «denostada» sentencia `ir_a (goto)`, *cuyo uso no se recomienda, pero sí el conocimiento de su funcionamiento.*

Capítulo 5. ***Flujo de control II: estructuras repetitivas.*** El capítulo introduce las estructuras repetitivas `(mientras ("while")`, `hacer-mientras ("do-while")`, `repetir ("repeat")`, `desde/para ("for"`, `interrumpir (break)`, `continuar (continue)`. Examina la repetición (*iteración*) de sentencias en detalle y compara los bucles controlados por centinela, bandera, etc. Explica precauciones y reglas de uso de diseño de bucles; compara los tres diferentes tipos de bucles, así como el concepto de bucles anidados.

Capítulo 6. ***Subprogramas (subalgoritmos): funciones.*** La resolución de problemas complejos se facilita considerablemente si se divide en problemas más pequeños (subproblemas). La resolución de estos problemas se realiza con *subalgoritmos* (subprogramas), que a su vez se dividen en dos grandes categorías: *funciones* y *procedimientos.*

Parte II. Estructura de datos

Esta parte es clave en el aprendizaje de técnicas de programación. Tal es su importancia que los planes de estudio de cualquier carrera de ingeniería informática o de ciencias de la computación incluye una asignatura troncal denominada *Estructura de datos.*

Capítulo 7. ***Estructuras de datos I (arrays –arreglos– y estructuras).*** Examina la estructuración de los datos en *arrays* o grupos de elementos dato del mismo tipo. El capítulo presenta numerosos ejemplos de *arrays* de uno, dos o múltiples índices. También se explican los otros tipos de estructuras de datos básicas: estructuras y registros. Estas estructuras de datos permiten encapsular en un tipo de dato definido por el usuario otros datos heterogéneos. Asimismo, se describe el concepto de arrays de estructuras y arrays de registros.

Capítulo 8. ***Las cadenas de caracteres.*** Se examina el concepto de carácter y de cadena `(String)` junto con su declaración e inicialización. Se introducen conceptos básicos de manipulación de cadenas: lectura y asignación junto con operaciones básicas, como longitud, concatenación, comparación, conversión y búsqueda de caracteres y cadenas. Las operaciones de tratamiento de caracteres y cadenas son operaciones muy usuales en todo tipo de programas.

Cristian Pilapanta 1095211-2202082300

Capítulo 9. ***Archivos (ficheros).*** El concepto de archivo junto con su definición e implementación es motivo de estudio en este capítulo. Los tipos de archivos más usuales junto con las operaciones básicas de manipulación se estudian con detenimiento.

Capítulo 10. ***Ordenación, búsqueda e intercalación.*** Las computadoras utilizan gran parte de su tiempo en operaciones de búsqueda, clasificación y mezcla de datos. Los archivos se sitúan adecuadamente en dispositivos de almacenamiento externo que son más lentos que la memoria central, pero que, por el contrario, tienen la ventaja de almacenamiento permanente después de apagar la computadora. Se describen los algoritmos de los métodos más utilizados en el diseño e implementación de programas.

Capítulo 11. ***Ordenación, búsqueda y fusión externa (archivos).*** Normalmente los datos almacenados de modo permanente en dispositivos externos requieren para su procesamiento el almacenamiento en la memoria central. Por esta circunstancia, las técnicas de ordenación y búsqueda sobre *arrays* y vectores comentados en el capítulo anterior necesitan una profundización en cuanto a técnicas y métodos. Una de las técnicas más importantes es la fusión o mezcla. En el capítulo se describen las técnicas de manipulación externa de archivos.

Capítulo 12. ***Estructuras dinámicas lineales de datos (pilas, colas y listas enlazadas).*** Una lista enlazada es una estructura de datos que mantiene una colección de elementos, pero el número de ellos no se conoce por anticipado o varía en un amplio rango. La lista enlazada se compone de elemento que contienen un valor y un puntero. El capítulo describe los fundamentos teóricos y las operaciones que se pueden realizar en la lista enlazada. También se describen los distintos tipos de listas enlazadas, tales como doblemente enlazadas y circulares. Las ideas abstractas de pila y cola se describen en el capítulo. Pilas y colas se pueden implementar de diferentes maneras, bien con vectores (arrays) o con listas enlazadas.

Capítulo 13. ***Estructuras de datos no lineales (árboles y grafos).*** Los árboles son otro tipo de estructura de datos dinámica y no lineal. Se estudian las operaciones básicas en los árboles junto con sus operaciones fundamentales.

Capítulo 14. ***Recursividad.*** El importante concepto de recursividad (propiedad de una función de llamarse a sí misma) se introduce en el capítulo junto con algoritmos complejos de ordenación y búsqueda en los que además se estudia su eficiencia.

Cristian Pilapanta 1095211-2202082301

Parte III. Programación orientada a objetos y UML 2.5.1

Nuestra experiencia en la enseñanza de la programación orientada a objetos a estudiantes universitarios data de finales de la década de los ochenta. En este largo periodo, los primitivos y básicos conceptos de orientación a objetos se siguen manteniendo desde el punto de vista conceptual y práctico, tal y como se definieron hace treinta años. Hoy, la programación orientada a objetos es una clara realidad y por ello cualquier curso de introducción a la programación aconseja, al menos, incluir un pequeño curso de orientación a objetos que puede impartirse como un curso independiente, como complemento de la parte II o como parte de un curso completo de introducción a la programación que comienza en el capítulo 1.

Capítulo 15. ***Tipos abstractos de datos, objetos y modelado con UML 2.5.1.*** Este capítulo describe los conceptos fundamentales de la orientación a objetos: clases, objetos y herencia. La definición y declaración de una clase junto con su organización y estructura se explican detenidamente. Se describen también otros conceptos importantes, tales como polimorfismo, ligadura dinámica y sobrecarga y un resumen de la terminología orientada a objetos.

Capítulo 16. ***Diseño de clases y objetos: representaciones gráficas en UML.*** Una de las tareas fundamentales de un programador es el diseño y posterior implementación de una clase y de un objeto en un lenguaje de programación. Para realizar esta tarea con eficiencia se exige el uso de una herramienta gráfica. UML es el lenguaje de modelado unificado estándar en el campo de la ingeniería de software, y en el capítulo se describen las notaciones gráficas básicas de clases y objetos.

Capítulo 17. ***Relaciones entre clases: delegaciones, asociaciones, agregaciones, herencia.*** En este capítulo se introducen los conceptos fundamentales de las relaciones entre clases: asociación, agregación y generalización/especialización. Se describen todas estas relaciones, así como las notaciones gráficas que las representan en el lenguaje de modelado UML.

Parte IV. Metodología de la programación y desarrollo de software

En esta parte se describen reglas prácticas para la resolución de problemas mediante programación y su posterior desarrollo de software. Estas reglas buscan proporcionar al lector reglas de puesta a punto de programas junto con directrices de metodología de programación que faciliten al lector la tarea de diseñar y construir programas con calidad y eficiencia junto con una introducción a la ingeniería de software.

Capítulo 18. ***Ingeniería de software y metodología de la programación.*** En el capítulo se analiza el desarrollo de un programa y sus diferentes fases: análisis, diseño, codificación, depuración, pruebas y mantenimiento. Estos principios básicos configuran la ingeniería de software como ciencia que pretende la concepción, diseño y construcción de programas eficientes.

Bibliografía y recursos de programación

Libros, revistas, web, EID, compiladores. Enumeración de los libros más sobresalientes utilizados por el autor en la escritura de esta obra, así como otras obras importantes de referencia que ayuden al lector que desee profundizar o ampliar aquellos conceptos que considere necesario conocer con más detenimiento. Listado de sitios web de interés para la formación en algoritmos, C/C++, Java, Python, C#, tanto profesionales como medios de comunicación, especialmente revistas especializadas.

Apéndices y guías de referencia (disponibles en sitio web www.mhhe.com/latam/joyanes.fp5e)

En todos los libros dedicados a la enseñanza y aprendizaje de técnicas de programación es frecuente incluir apéndices de temas complementarios a los explicados en los capítulos anteriores. Estos apéndices sirven de guía y referencia de elementos importantes del lenguaje y de la programación de computadoras.

Apéndice A. ***Representación de la información en las computadoras.***

Apéndice B. ***Prioridad de operadores.*** Tabla que contiene todos los operadores y el orden de prioridad y asociatividad en las operaciones cuando aparecen en expresiones.

Apéndice C. ***Código ASCII y Unicode.*** Tablas de los códigos de caracteres que se utilizan en programas de computadoras. El código ASCII es el más universal y utilizado de modo masivo por programadores de todo el mundo y, naturalmente, es el que utilizan la mayoría de las computadoras actuales. Unicode es un lenguaje mucho más amplio que utilizan las computadoras personales para realizar programas y aplicaciones en cualquier tipo de computadora y en Internet. Unicode proporciona un número único para cada carácter, sin importar la plataforma, sin importar el programa, sin importar el idioma. La importancia de Unicode reside, entre otras cosas, en que está avalado por líderes de la industria tales como Apple, HP, IBM, Microsoft, Oracle, entre otros. También es un requisito para los estándares modernos, como XML, Java, C#, etcétera.

Apéndice D. ***Palabras reservadas de Java, C++, C, C#, R y Python.*** Se presentan las versiones más utilizadas de los diferentes lenguajes de programación: Java, C, C++, Python, Java Script y R.

Apéndice E. ***Índice TIOBE de lenguajes de programación.***

Apéndices y guías de sintaxis en el sitio web (www.mhhe.com/latam/joyanes_fp5e)

Apéndice I. ***Guía de sintaxis del lenguaje ANSI C.*** Especificaciones, normas de uso y reglas de sintaxis del lenguaje de programación C en su versión estándar ANSI/ISO.

Cristian Pilapanta 1096211-2202082303

Apéndice II. ***Guía de sintaxis del lenguaje ANSI/ISO C++ estándar.*** Especificaciones, normas de uso y reglas de sintaxis del lenguaje de programación C++ en su versión estándar ANSI/ISO.

Apéndice III. ***Guía de sintaxis del lenguaje Java.*** Especificaciones, normas de uso y reglas de sintaxis del lenguaje de programación Java en su versión estándar ANSI/ISO.

Apéndice IV. ***Guía de sintaxis del lenguaje Python.*** Especificaciones, normas de uso y reglas de sintaxis del lenguaje de programación Python, versión 3.0.

Apéndice V. ***Especificaciones del lenguaje algorítmico UPSAM 2.5***

Cristian Pilapanta 1095211-2202082304

Agradecimientos a la quinta edición

A lo largo de muchos años, un gran número de personas ha contribuido a la evolución de este libro y ha colaborado para que pudiese llegar a la quinta edición, desde el ya remoto 2008, cuando se publicó la cuarta edición y en la década anterior, las tres primeas ediciones, así como en la inicial *Metodología de la Programación* que constituyó el embrión y los cimientos de *Fundamentos de Programación*.

Quiero manifestar expresamente mi eterno agradecimiento a la infinidad de personas (estudiantes, profesores, maestros, colegas del ámbito académico y profesional, autodidactas de la programación, revisores y todo el universo de lectores) que han comentado, revisado, corregido, criticado y examinado los contenidos de las ediciones anteriores y nos han dado sugerencias y propuestas para esta nueva edición. Por esta razón he querido mantener expresamente los agradecimientos de la cuarta edición, y de nuevo reconocerles mi eterno agradecimiento, valga la redundancia, por su contribución a dicha edición y que han influido considerablemente en la redacción de la nueva quinta edición.

Al igual que en todas mis obras, siempre he de agradecer al equipo editorial de McGraw-Hill de las sedes de Madrid –Cristina García, Editorial Project Manager España y México (CDMX) – Guillermo Domínguez Chávez, Senior Editor LATAM y Gabriela López Ballesteros, Portfolio Manager LATAM, que tan bien me aconsejan, asesoran y recomiendan sobre los diferentes aspectos de la edición del libro y hasta que la obra llega al universo mundial de las librerías físicas y de la Web.

En Carchelejo (Sierra Mágina, Jaén, Andalucía), España
y en Ciudad de México (CDMX),
México, *febrero de 2020.*

Cristian Pliqueria 1095211-2200082307

PARTE I

ALGORITMOS Y HERRAMIENTAS DE PROGRAMACIÓN

Cristian Pliqueria 1096211-2202082306

1096211220208230

CONTENIDO

CAPÍTULO 1 Introducción a las computadoras y a los lenguajes de programación

CAPÍTULO 2 Metodología de la programación y desarrollo de software

CAPÍTULO 3 Estructura general de un programa

CAPÍTULO 4 Flujo de control I: estructuras selectivas

CAPÍTULO 5 Flujo de control II: estructuras repetitivas

CAPÍTULO 6 Subprogramas (subalgoritmos): funciones

1095211220208230

CAPÍTULO 1

Introducción a las computadoras y a los lenguajes de programación

Cristian Pilapanta 1006211-2202082310

1.1. El origen de las computadoras y su evolución
1.2. Las computadoras modernas: una breve taxonomía
1.3. Organización de una computadora
1.4. Software: conceptos básicos y clasificación
1.5. Sistema operativo
1.6. El lenguaje de la computadora
1.7. La programación de las computadoras en perspectiva
1.8. Lenguajes de programación
1.9. Traductores de lenguaje: el proceso de traducción de un programa
1.10. Evolución de los lenguajes de programación
1.11. Paradigmas de programación
1.12. Internet y la Web
1.13. *Cloud computing* (computación en la nube como servicio)
1.14. Internet de las cosas
1.15. Big Data. Los grandes volúmenes de datos
1.16. Los lenguajes de programación más populares: índice TIOBE
1.17. Nacimiento de la programación moderna: lenguajes de programación de referencia (C, C++, Java, Python y C#)
RESUMEN

INTRODUCCIÓN

Las computadoras (u ordenadores) son dispositivos (máquinas) esenciales en la vida diaria de las personas, tanto en sus actividades domésticas como en las profesionales o de ocio. Estas máquinas procesan datos bajo el control de una secuencia de instrucciones llamada **programa de computadora**, incluida en la disciplina general denominada *software*. Los programas conducen a las computadoras mediante un conjunto de acciones especificadas que han sido diseñadas y escritas, previamente, por los profesionales conocidos como **programadores** de computación. Los programas de computadora contienen una lista de instrucciones escritas por los programadores en un **lenguaje de programación** y traducidas posteriormente a un lenguaje "comprensible" por las máquinas, **lenguaje máquina**.

Una computadora tiene una organización o estructura que consta de diferentes unidades o dispositivos físicos conocidos como hardware (p. ej. procesadores, teclados, monitores o pantallas, discos duros, ratones [mouses], etcétera) y unidades lógicas conocidas como software (sistemas operativos, lenguajes de programación, compiladores, intérpretes, aplicaciones...). Los programas de computadora deben ser escritos en un lenguaje de programación (C, C++, Java, Python, R, C#...) que se traducirá, posteriormente, a lenguaje máquina para que la computadora lo pueda interpretar.

En este capítulo se dará una introducción a la organización física y lógica de las computadoras (hardware y software), y a los lenguajes de programación (conceptos, estructuras, categorías, historia y evolución). Después, en el capítulo 2 y posteriores se

describirán los conceptos fundamentales de los algoritmos (instrucciones paso a paso para la resolución de un determinado problema), soporte previo que han de diseñar los programadores para, después, convertirlos en programas escritos en un lenguaje de programación.

1.1. El origen de las computadoras y su evolución

Todas las historias publicadas sobre el nacimiento de las computadoras[1] comienzan remontándose a la antigüedad con la creación del ábaco como primera máquina de cálculo. Este instrumento, inventado en Asia, se utilizó en la antigua Babilonia, en China y, por supuesto, en Europa. Actualmente se utiliza con fines educativos y de ocio.

En 1642, el filósofo y matemático francés Blas Pascal inventó la primera calculadora mecánica, conocida como **pascalina**. La calculadora tenía una serie de engranajes (ruedas dentadas) que permitían realizar sumas y restas, con un método entonces ingenioso y revolucionario: cuando se giraban los dientes de la primera rueda, avanzaba un diente de la segunda rueda, al girar los dientes de la segunda rueda, avanzaba un diente de la tercera, y así sucesivamente. Tanto el ábaco como la pascalina sólo podían realizar sumas y restas. Asimismo, a finales del siglo XVII, en 1694, el científico alemán **Gottfried Leibniz** inventó una máquina que podía sumar, restar, multiplicar y dividir. En 1819, el francés **Joseph Jacquard** creó las bases de las tarjetas perforadas como soporte de información.

Generaciones de computadoras

Sin embargo, casi todos los historiadores coinciden en que la era de partida de la computación moderna fue a principios del siglo XIX. El físico y matemático inglés **Charles Babbage** construyó dos máquinas calculadoras: la máquina de diferencias y la máquina analítica. La primera podía realizar automáticamente operaciones complejas como elevar números al cuadrado. Babbage construyó un prototipo de la máquina diferencial, aunque nunca fue fabricado para producción. Entre 1833 y 1835 diseñó la máquina analítica, una calculadora que incluía un dispositivo de entrada, dispositivo de almacenamiento de memoria, una unidad de control que permitía instrucciones de proceso en secuencias y dispositivos de salida. Esta máquina sentó las bases de la computación moderna, aunque los trabajos de Babbage fueron publicados por su colega **Ada Augusta**, **condesa de Lovelace**, considerada como la primera programadora de computadoras del mundo. A finales del siglo XIX se utilizaron por primera vez con éxito las tarjetas perforadas que ayudaron a realizar el censo de Estados Unidos de América en 1890. **Herman Hollerith** inventó una máquina calculadora que funcionaba con electricidad y utilizaba tarjetas perforadas para almacenar datos. La máquina de Hollerith tuvo un gran éxito y apoyándose en ella creó la empresa Tabulating Machine Company, que más tarde se convirtió en la reconocida **IBM**.

Se considera que Mark I fue la primera computadora digital de la historia que aprovechó el éxito de la máquina de tarjetas perforadas de Hollerith. IBM construyó **Mark I** en 1944 junto con la Universidad de Harvard y con la dirección de **Howard Aiken**. Las entradas y salidas de datos se realizaban mediante tarjetas y cintas perforadas. En 1939, **John V. Atanasoff**, profesor de la Universidad de Iowa y el estudiante de doctorado **Clifford E. Berty**, construyeron un prototipo de la ABC, una computadora digital que utilizaba tubos de vacío (válvulas) y el sistema de numeración digital de base 2, además de que disponía de unidad de memoria. La computadora no se comercializó y el proyecto fue abandonado, pero en 1973, un tribunal federal de Estados Unidos de América reconoció de modo oficial a Atanasoff los derechos sobre la invención de la computadora digital electrónica automática. Entre 1942 y 1946, **John P. Eckert**, **John W. Marchly** y su equipo de la Universidad de Pensilvania, construyeron la **ENIAC**, considerada la primera computadora digital de la historia y que fue utilizada desde 1946 hasta 1955 (contenía 18 000 tubos de vacío y pesaba 30 toneladas).

[1] En España está muy extendido el término **ordenador** para referirse a la traducción de la palabra inglesa *computer*. El DRAE (*Diccionario de la lengua española*, de la Real Academia Española y todas las Academias de la Lengua de Latinoamérica, África y Asia) acepta, indistintamente, los términos sinónimos: **computador**, **computadora** y **ordenador**. Entre las diferentes acepciones define la computadora electrónica como: "máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos". En el *Diccionario panhispánico de dudas* (Madrid: RAE, 2005, p. 157), editado también por la Real Academia Española y la Asociación de Academias de la Lengua Española, se señala que el término **computadora** (del término inglés *computer*) se utiliza en la mayoría de los países de América, mientras que el masculino **computador** es de uso mayoritario en Chile y Colombia; en España se usa preferentemente el término **ordenador**, tomado del francés ***ordinateur***. En este joven diccionario la definición de computador es: "máquina electrónica capaz de realizar un tratamiento automático de la información y de resolver con gran rapidez problemas matemáticos y lógicos mediante programas informáticos".

Cristian Pilpanta 1006211-2020060010

Las computadoras como se les conoce hoy día siguen el **modelo von Newmann**. El matemático **John von Newmann** realizó un estudio teórico a finales de la década de 1940, en los que sentó las bases de la organización y reglas de funcionamiento de la computadora moderna. Su diseño incluía componentes como unidad lógica y aritmética, unidad de control, unidad de memoria y dispositivos de entrada/salida. Estos componentes se describen en el apartado siguiente. Desde el punto de vista comercial, las computadoras más conocidas y que aparecieron en las décadas de 1950 y 1960 son: Univac I, IBM 650, Honeywell 400 y las populares IBM 360.

El **transistor** fue inventado en 1948 como sustituto de la válvula o tubo de vacío, y los transistores comenzaron a aparecer en las computadoras ocho años más tarde. Las computadoras que utilizaban transistores fueron radicalmente más pequeñas, más fiables y más económicas que las válvulas de vacío para almacenar y manipular datos. A finales de la década de 1950, los investigadores desarrollaron el **circuito integrado**, un pequeño **chip de silicio** que contenía centenares y millares de transistores y otros dispositivos electrónicos. A mediados de la década de 1960, las computadoras basadas en transistores fueron reemplazadas por otras más pequeñas, pero más potentes, construidas alrededor de estos nuevos circuitos integrados. Los circuitos integrados reemplazaron a los transistores por las mismas razones que éstos reemplazaron a las válvulas de vacío: *fiabilidad, tamaño, velocidad, eficiencia* y *costo*. Las invenciones anteriores produjeron un gran impacto en la sociedad, pero fue el invento desarrollado por Intel, en 1971, el primer microprocesador, un único chip de silicio, que contenía todos los componentes de una computadora.

Esta era se destacó también por la aparición de la industria de desarrollo de software con la introducción de los dos lenguajes de programación de alto nivel: **Fortran** (1954, aplicaciones científicas) y **Cobol** (1959, aplicaciones de negocios). Estos dos lenguajes de programación de alto nivel fueron reemplazando en el desarrollo al lenguaje ensamblador que a su vez sustituyó al lenguaje máquina (basado en ceros y unos), que es el lenguaje con el que funcionan las computadoras.

La revolución de las computadoras personales comenzó a finales de la década de 1970 cuando Apple, Commodore y Tandy, y otras compañías, introdujeron computadoras de bajo costo, basadas en microprocesadores que ocupaban muy poco espacio físico. En 1981, IBM presentó su **computadora personal (PC, Personal Computer)** y junto con Microsoft, creador del sistema operativo MS-DOS que sustentaba a esta computadora, fueron las compañías que iniciaron la nueva era de la computación que todavía hoy vivimos y disfrutamos.

Las computadoras de hoy día son muy potentes, fiables y fáciles de utilizar. Se han convertido en herramientas indispensables en la vida diaria, lo mismo para los niños en las escuelas que para los profesionales en las empresas. Ellas pueden aceptar instrucciones orales (de voz) e imitar el razonamiento humano mediante técnicas de inteligencia artificial. En la actualidad, las computadoras se utilizan para ayudar a los médicos en el diagnóstico de enfermedades, a los empresarios en la toma de decisiones y a los militares a posicionarse geográficamente en el terreno mediante satélites y sistemas GPS. Las computadoras portátiles crecen en números exponenciales y las *netbooks* y tabletas (computadoras de 7 a 11 pulgadas de tamaño de pantalla) y los teléfonos inteligentes (*smartphones*) se están convirtiendo en el elemento de cálculo y de acceso a Internet por antonomasia.

Los teléfonos celulares (móviles) y las redes de comunicaciones móviles y fijas, redes inalámbricas, sin cables, están configurando un nuevo modelo de computación, computación móvil o computación celular, que está trayendo infinidad de aplicaciones de computación móvil. Los dispositivos portátiles (*handheld*) permiten conexiones a Internet, envío de correos electrónicos (*e-mail*), conexión a redes sociales, navegar con sistemas GPS de posicionamiento global, visualizar mapas de ciudades, de carreteras, etcétera.

En lo relativo a los lenguajes de programación utilizados por las computadoras hay que destacar que cada computadora procesa instrucciones en un lenguaje nativo denominado **lenguaje máquina**. El lenguaje máquina utiliza códigos numéricos para representar las operaciones básicas de la computadora: sumas, restas, productos, etc. Este lenguaje, aunque era tedioso, resultaba imprescindible para el funcionamiento de la máquina; sin embargo, poco a poco fueron apareciendo otros lenguajes, primero el **ensamblador**, que ya utilizaba palabras cortas para identificar a las instrucciones y por último los **lenguajes de alto nivel**, ya más similares a los humanos. Los lenguajes de programación de alto nivel generalistas (**C**, **C++**, **Java**, **Python**, **C♯**, **R**, **Scala**, **Swift**, **Ruby**) o especializados en el desarrollo de la Web como **HTML**, **XML**, **JavaScript**.

1.2. Las computadoras modernas: una breve taxonomía

Hoy día la computación se está volviendo cada vez más ubicua, de modo que se puede leer un correo electrónico, escuchar una canción, hacer una reservación para el avión o el tren, acceder a las cuentas de su banco, reproducir

Cristian Pulgarín 1085211-2202060010

un video o ver una película desde cualquier lugar, en cualquier dispositivo y en cualquier momento. Desde el punto de vista técnico, aunque existen diferentes categorías de computadoras, por ahora, todas ellas tienen la misma organización, comparten los mismos elementos y siguen el modelo de Von Newmann, aunque la computadora cuántica cada vez se encuentra en un estado más avanzado de desarrollo. Si bien, todas ellas comparten los mismos elementos básicos, existen diferentes categorías de computadoras dependiendo de su tamaño y potencia. La clasificación más común es:

- **Grandes computadoras** (*mainframes*). Computadoras enfocadas, esencialmente, a la resolución de programas complejos y especializados de grandes organizaciones y empresas.
- **Supercomputadoras.** Grandes computadoras dedicadas especialmente a la investigación y construidas con decenas y cientos de miles de microprocesadores o múltiples núcleos. Son las computadoras utilizadas por excelencia para investigación y desarrollos avanzados en i+D+i.
- **Servidores.** Un servidor es una computadora que proporciona a otras computadoras conectadas en red el acceso a datos, programas u otros recursos. Existen numerosos tipos de servidores, por ejemplo, los servidores web responden a las peticiones en páginas web, los servidores de bases de datos manejan las consultas en las bases de datos, servidores de impresión proporcionan acceso a impresoras; servidores de correo dan acceso al correo electrónico, etcétera. Aunque cualquier computadora de escritorio podría actuar como servidor, las computadoras servidor se fabrican expresamente con estas finalidades. Los servidores pueden tener procesadores más rápidos, más memoria y conexiones más veloces que los sistemas de escritorio típicos. Los servidores se pueden agrupar en *clusters* con el objeto de aumentar su potencia de procesamiento.
- **Computadoras personales y estaciones de trabajo.** Una computadora personal (PC) está, por lo general, diseñada para ser utilizada por una persona en un momento dado y es una herramienta de productividad, creatividad o comunicación. Las PC pueden clasificarse en computadoras de escritorio, estaciones de trabajo, portátiles (*laptops, notebooks*...), videoconsolas.
- **Computadoras móviles.** Computadoras o dispositivos que se pueden transportar fácilmente y que sirven para cubrir las necesidades de los usuarios, normalmente, que están en movimiento. Estas computadoras son esencialmente teléfonos inteligentes, tabletas (*tablets*), videoconsolas, etcétera.
- **Sistemas embebidos.** Más de 90% de los procesadores existentes en el mundo se ocultan o "embeben" (empotran) en dispositivos electrónicos, como sensores y en los más variados aparatos, como automóviles, aviones, trenes, barcos. Un procesador utilizado como componente de un sistema físico mayor se llama sistema embebido. Se puede encontrar este tipo de sistemas, además de en los aparatos citados, en luces de tráfico, juguetes, máquinas de juegos, aparatos de TV, tarjetas inteligentes y sensores más diversos. Los procesadores en sistemas embebidos actúan de modo similar a como lo hacen en computadoras personales. Cuando un programa se incrusta en un chip se le conoce como firmware.

Las líneas de separación de las categorías anteriores no siempre están bien definidas; así, por ejemplo, un reloj inteligente se puede considerar un dispositivo móvil y un sistema embebido en las grandes computadoras o supercomputadoras que se utilizan para aplicaciones de computación intensivas, tales como cálculos meteorológicos, modelado biológico avanzado o investigación genérica. Las supercomputadoras pueden operar con velocidades de cuatrillones (10^{15}) de cálculos por segundo. Existen desarrollos de computadoras que pueden operar en quintillones (10^{18}) operaciones por segundo. Se están desarrollando en diferentes países del mundo. En España, el Centro de Supercomputación de Barcelona (**BSC**, Barcelona Supercomputing Center) es referencia nacional y europea y con grandes referencias a nivel mundial.

Las computadoras *mainframe* son utilizadas por las empresas para aplicaciones específicas como procesamiento de datos (*data warehouses* y **Big Data**). Sistemas clásicos son los sistemas de emisión de billetes y tarjetas de embarque en aviones de las compañías aéreas, así como los sistemas para administrar transacciones financieras.

Las computadoras personales son las computadoras de escritorio y las portátiles (**laptops** y **notebooks**). Existen diferentes tamaños y con características muy diversas. Las computadoras o dispositivos móviles inteligentes incluyen dispositivos de computación portátiles como teléfonos inteligentes, lectores de libros electrónicos (*e-readers*), tabletas, etcétera. Éstos se utilizan normalmente para realizar llamadas telefónicas, enviar mensajes de texto, **chats**, videoconferencias, asistentes virtuales o navegar en páginas web.

Cristian Pilapanta 1005211-2020000010

1.3. Organización de una computadora

No es necesario conocer todos los detalles del funcionamiento de una computadora para poder trabajar con ella, pero la comprensión de sus principios fundamentales le ayudará considerablemente a desarrollar y poner en acción sus programas. Es como la conducción de un automóvil (carro o coche): el conocimiento de cómo funciona el proceso de combustión interna de los motores o el sistema de frenado le ayuda a entender cómo funcionan componentes o servicios, tales como el depósito de combustible, arranque del motor o el funcionamiento del acelerador o el freno.

Aunque las diferentes computadoras pueden variar significativamente en detalles específicos, en una visión amplia, todas las computadoras digitales modernas son muy similares. Una vista funcional o la organización y estructura de una computadora se muestra en la figura 1.1.

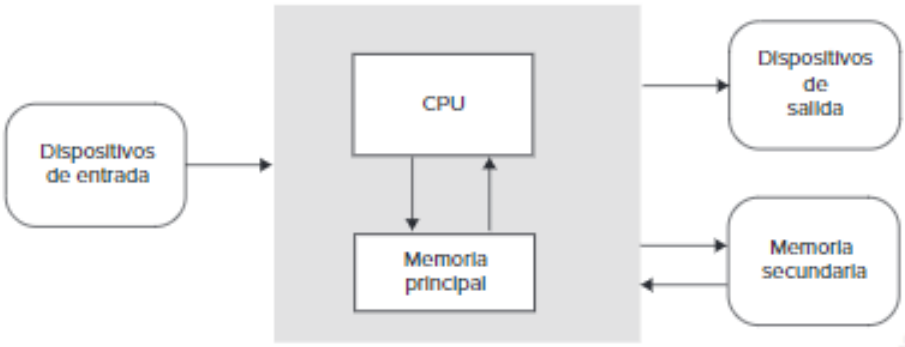


Figura 1.1. Organización de una computadora.

Cristian P Espana 1096211-2202080016

El procesador obtiene instrucciones y datos de la memoria. Los dispositivos de entrada escriben datos en la memoria; los dispositivos de salida leen datos de la memoria; el procesador a través de su unidad de control envía las señales que determina el funcionamiento de los caminos de datos, memoria, entrada y salida.

Los componentes principales de una computadora son:

1. Unidad Central de Procesamiento (**UCP**; **CPU**, *central processing unit*) o procesador para interpretar y ejecutar programas.
2. Memoria, para almacenar datos y programas (principal y secundaria).
3. Dispositivos de entrada.
4. Dispositivos de salida.
5. Sistema de comunicación para transferir datos a y desde el mundo exterior.

Procesador (CPU/UCP)

El **procesador** (**CPU**) consta de una Unidad Aritmética Lógica (ALU) para realizar cálculos y tomar decisiones, y una Unidad de Control para actuar dirigiendo datos a las posiciones correctas. Cuando un programa se ejecuta, la **CPU** ejecuta las instrucciones encontradas en la memoria. La CPU es el "cerebro" de la máquina y se denomina también procesador. En la CPU se realizan todas las operaciones básicas de la computadora. La CPU puede realizar operaciones como sumar dos números y también operaciones lógicas como comprobar si dos números son iguales.

Memoria

La **memoria** se utiliza para almacenar todo lo que necesita la computadora. Existen dos tipos: 1) memoria de larga duración, tal como unidades de disco y unidades *flash* que almacenan datos cuando la computadora está apagada; 2) memoria temporal que pierde los datos cuando se desconecta la alimentación eléctrica e incluye registros y me-

moria RAM. La memoria es el espacio que necesita el procesador para almacenar información que está actualmente funcionado y es una memoria a largo plazo para almacenar los datos necesarios en el futuro. La memoria es jerárquica; por ejemplo, una unidad de disco duro proporciona un lugar para almacenar datos a largo plazo, mientras que la memoria caché es una memoria pequeña pero rápida y cara, que almacena los datos que se utilizan con más frecuencia. Accediendo a la caché, la CPU puede leer y escribir datos rápidamente, sin necesidad de recurrir al sistema completo. La memoria almacena programas y datos. La CPU puede acceder directamente sólo a la información que se almacena en la memoria principal, memoria de acceso directo. La memoria principal es rápida, pero también es volátil; esto es, cuando se apaga la energía, la información en la memoria se pierde. Por consiguiente, debe existir una memoria secundaria que proporcione capacidad de almacenamiento de modo permanente.

En una computadora personal moderna, la memoria secundaria normalmente consta de una unidad de disco duro (HDD, *hard disk drive)* y/o una unidad de estado sólido (SSD, *solid state drive*). Un HDD almacena información como patrones magnéticos en un disco giratorio, mientras que un SSD emplea circuitos electrónicos conocidos como memoria *flash.* La mayoría de las computadoras también soportan medios intercambiables o removibles de memoria secundaria tales como memorias USB (pendrive, un formato de memoria flash) y DVD (disco versátil digital), cada vez más en desuso, que almacenan información como patrones ópticos que se leen y escriben mediante un láser.

Dispositivos de entrada y salida

Los dispositivos de E/S (Entrada/Salida) son dispositivos que permiten comunicarse con la computadora tales como teclados, ratones, monitores, impresoras, cámaras web, teléfonos inteligentes, escáneres, tabletas y otras unidades de almacenamiento.

Las personas interactúan con las computadoras a través de dispositivos de entrada y salida familiares como teclados, ratones y monitores (pantallas de video). La información de los dispositivos de entrada se procesa en la CPU y se envía a la memoria principal o secundaria. De modo similar cuando la información necesita ser visualizada, la CPU la envía a uno o más de los dispositivos de salida; por ejemplo, cuando se ejecuta una aplicación de juegos, una hoja de cálculo o un tratamiento de textos. Primero, las instrucciones que contiene el programa se copian de la memoria secundaria permanente a la memoria principal; una vez que se cargan las instrucciones la CPU comienza a ejecutar el programa. Técnicamente la CPU realiza un proceso denominado ciclo de extracción-ejecución (*fetch-execute*). La primera instrucción se recupera de la memoria, se decodifica para ver su significado y se ejecuta; a continuación, se extrae la siguiente instrucción, se decodifica y se ejecuta; el ciclo continúa, instrucción a instrucción. En esencia, éstas son las actividades que realiza la computadora desde que se enciende hasta que se apaga de nuevo: extraer, decodificar, ejecutar. La computadora ejecuta un flujo de instrucciones sencillas o miles de millones de instrucciones cada segundo.

Características típicas de computadoras portátiles

Gama baja	
Memoria central RAM	4 a 8 GB
SSD	128 GB
Procesador	I3 de Intel
Gama media	
Memoria central RAM	8 a 16 GB
SSD	128-256 GB
Procesador	I5 de Intel
Disco duro (HDD)	512 GB y 1TB
Gama alta	
Memoria central RAM	16 a 32 GB
SSD	256 GB
Procesador	I7 de Intel
Disco duro (HDD)	1 TB y 2TB

Cristian Pilapanta 1006211-2020/00017

1.3.1. Memoria central de la computadora

El trabajo principal de la CPU es seguir las instrucciones codificadas de los programas, pero sólo puede manejar una instrucción (aunque ya existen hoy día, como veremos más adelante, las computadoras multiprocesadores que pueden efectuar varios procesos simultáneamente) y unos pocos datos en cada momento. La computadora necesita un espacio para almacenar el resto del programa y los datos hasta que el procesador está listo para utilizar; ésta es la tarea que realiza la memoria RAM.

La **memoria RAM** o simplemente **memoria** es el tipo más importante de almacenamiento principal de la computadora. Los chips de RAM contienen circuitos que almacenan las instrucciones de los programas, los datos de entrada y los resultados temporalmente.

En la mayoría de las computadoras existen dos tipos de memoria principal: **memoria de acceso aleatorio**, **RAM**, que soporta almacenamiento temporal de programas y datos, y **memoria de sólo lectura**, **ROM**, que almacena datos o programas de modo permanente.

En general, la información almacenada en la memoria puede ser de dos tipos: las *instrucciones* de un programa y los *datos* con los que operan las instrucciones. Para que un programa se pueda *ejecutar* (correr, rodar, funcionar..., en inglés *run*), debe estar situado en la memoria central, en una operación denominada carga (*load*) del programa. Después, cuando se ejecuta el programa, *cualquier dato a procesar por el programa se debe llevar a la memoria* mediante las instrucciones del programa. En la memoria central, hay también datos diversos y espacio de almacenamiento temporal que necesita el programa cuando se ejecuta, a fin de poder funcionar.

La memoria RAM es un tipo de memoria volátil (su contenido se pierde cuando se apaga la computadora); esta memoria es, en realidad, la que se suele conocer como memoria principal o de trabajo; en esta memoria se pueden escribir datos y leer de ella. Esta memoria RAM puede ser *estática* (**SRAM**) o *dinámica* (**DRAM**) según sea el proceso de fabricación. Las memorias RAM actuales más utilizadas son: **SDRAM** en sus tres tipos: **DDR** (double data rate), **DDR2** y **DDR3**.

Cristian Pilapanta 1095211-2202090018

Ejecución

Cuando un programa corre o funciona en una computadora se dice que se ejecuta.

En la memoria principal se almacenan:

- Los datos enviados *para* procesarse desde los dispositivos de entrada.
- Los programas que realizarán los procesos.
- Los resultados obtenidos preparados para enviarse a un dispositivo de salida.

La memoria ROM es una memoria que almacena información de modo permanente en la que no se puede escribir (viene pregrabada, "grabada", por el fabricante), ya que es una memoria de sólo lectura. Los programas almacenados en ROM no se pierden al apagar la computadora y cuando se enciende (se prende), se lee la información almacenada en esta memoria. Al ser esta memoria de sólo lectura, los programas almacenados en los chips ROM no se pueden modificar y suelen utilizarse para almacenar los programas básicos que sirven para arrancar la computadora.

Con el objetivo de que el procesador pueda obtener los datos de la memoria central más rápidamente, la mayoría de los procesadores actuales (muy rápidos) utilizan con frecuencia una memoria denominada *caché* que sirve para almacenamiento intermedio de datos entre el procesador y la memoria principal. La memoria *caché*, en la actualidad, se incorpora casi siempre al procesador.

Los programas y los datos se almacenan en RAM. Las memorias de una computadora personal se miden en unidades de memoria y suelen ser actualmente de 1 a 8 GB (gigabytes), aunque ya es frecuente encontrar memorias centrales de 16 y 32 GB en computadoras personales y en cantidad mayor en computadoras profesionales y en servidores.

Normalmente una computadora contiene mucha más memoria RAM que memoria ROM interna; también la cantidad de memoria se puede aumentar hasta un máximo especificado, mientras que la cantidad de memoria ROM normalmente es fija. Cuando en la jerga informática y en este texto se menciona la palabra memoria se suele referir a memoria RAM, que normalmente es la memoria accesible al programador.

La RAM es una memoria muy rápida y limitada en tamaño; sin embargo, la computadora tiene otro tipo de memoria denominada memoria secundaria o almacenamiento secundario que puede crecer comparativamente en términos mucho mayores. La memoria secundaria es realmente un dispositivo de almacenamiento masivo de información y por ello a veces se le conoce como memoria auxiliar, almacenamiento auxiliar, almacenamiento y memoria externos.

1.3.2. Memoria caché

Los procesadores de las computadoras son muy rápidos y están constantemente leyendo información de la memoria, lo que implica que, frecuentemente, deben esperar a que llegue la información ya que los tiempos de acceso a la memoria son más lentos que la velocidad del procesador. En consecuencia, surgió la necesidad de incorporar una memoria interna que fuera una memoria pequeña temporal pero rápida para que el procesador pudiera utilizarla para obtener información que necesitara de modo inmediato. Así nació la memoria caché.

Una memoria caché[2] es una memoria temporal, pequeña pero más rápida que el procesador y se utiliza para obtener información que se va a necesitar en momentos de inmediatez. La memoria caché es una memoria RAM estática situada físicamente entre el procesador y la memoria principal cuyo objetivo primordial es lograr que la memoria del procesador trabaje a su velocidad de procesamiento. Normalmente, el procesador solicita información de modo más rápido de lo que la memoria principal del sistema es capaz de ofrecer; por esta razón, se introduce el procesador en un estado "de espera" ya que, entonces, el procesador no hace nada hasta que no recibe la información solicitada. En la figura 1.2 Stalling (2016) muestra cómo la transferencia de bloques de la memoria principal se realiza a la CPU mediante transferencia de bloques de modo lento, a través de la memoria caché que, a su vez, ya envía transferencia de palabra de un modo más rápido a la CPU. La memoria caché se construye utilizando una tecnología de memoria estática de acceso aleatorio (SRAM) de muy alta velocidad, que se compone de un número de bloques de longitud fija de K palabras, y de menor capacidad, dependiendo de la categoría de la caché –niveles L1, L2 o L3– (del orden de KBytes, aunque en las computadoras modernas, la memoria caché puede llegar a MBytes e incluso GBytes); en la actualidad la mayoría de las computadoras portátiles incorporan memorias caché de tipo flash que pueden almacenar de 8 a 32 GB, lo que facilita considerablemente la velocidad, hoy día están prácticamente integradas en la memoria principal, que en el caso de disponer de memorias SSD puede llegar hasta 256 GB.

Cristian Pilapanta 1095211-2202090018

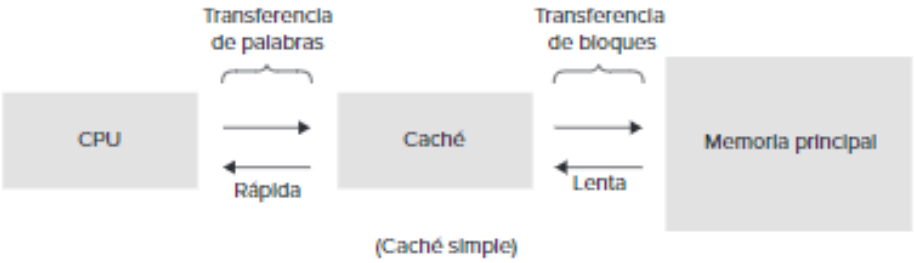


Figura 1.2. Arquitectura de una memoria caché.
Fuente: Stallings (2016: 128)

Memoria caché: cantidad pequeña de memoria rápida SRAM

- Integrada en el procesador.
- Más rápida y cara que la memoria principal DRAM.
- Está situada entre la memoria principal y el procesador.
- Almacena una copia de partes de la memoria principal.

[2] El término caché aparece en el DRAE como "Inform. *Memoria caché*" con un hiperenlace dirigido al término memoria. Allí entre las numerosas acepciones del término aparece memoria caché como: "memoria de acceso rápido de una computadora u otro dispositivo electrónico que guarda temporalmente las últimas informaciones procesadas".

1.3.3. Unidades de medida de memoria

La unidad básica de almacenamiento es el *byte*. Cada byte contiene una dirección única. La dirección se utiliza para localizar el byte de almacenamiento y recuperación de datos. A los bytes se puede acceder en cualquier orden, de modo aleatorio, y por esta razón la memoria principal se denomina RAM. Los múltiplos de la unidad de memoria básica son 1 **kilobyte**[3] (**K** o **Kb**) equivalente a 1.024 bytes, aunque desde el punto de vista práctico 1 Kb se suele considerar equivalente a 1.000 bytes; 1 **megabyte** (**M** o **Mb**) equivale a 1.024 × 1.024 bytes o de modo práctico, 1 millón de bytes; 1 **gigabyte** (**G** o **Gb**) equivale a 1.024 × 1.024 × 1.024 bytes, equivalente a 1.000 millones de bytes; 1 **terabyte** (**T** o **Tb**), equivalente a 1.024 gigabytes; 1 **petabyte** (**P** o **Pb**) equivale a 1.024 terabytes y 1 **exabyte** (**E** o **Eb**), equivalente a 1.024 petabytes. La tabla 1.1 muestra las unidades básicas de medida de almacenamiento en memoria.

Tabla 1.1. Unidades de medida de almacenamiento.

Unidad	Símbolo	Tamaño en bits/bytes	
Byte	B	8 bits	1 B = 8 bits
Kilobyte	KB	2^{10} bytes = 1.024 bytes	1 KB = 1.024 B
Megabyte	MB	2^{20} bytes = 1.048.576 bytes; 1.024 Kb = 2^{10} Kb	1 MB = 1.024 KB
Gigabyte	GB	2^{30} bytes = 1.073.741.824 bytes; 1.024 Mb = 2^{10} Mb = 2^{20} Kb	1 GB = 1.024 MB
Terabyte	TB	2^{40} bytes = 1.099.511.627.776 bytes 1 TB = 1.024 GB = 2^{10} GB = 2^{20}	MB = 230 KB
Petabyte	PB	2^{50} bytes = 1.125.899.906.842.624 bytes	TB = 2^{20} GB
Exabyte	EB	1 PB = 1.024 TB = 2^{10} 2^{60} bytes; 1.024 Pb = 2^{10} Pb = 2^{20} TB	1 EB = 1.024 PB
Zettabyte	ZB	2^{70} bytes; 1.024 Eb	1 ZB = 1.024 EB
Yotabytte	YB	2^{80} bytes; 1.024 Zb	1 YB = 1.024 ZB

Bit: un dígito binario 0 o 1.
Byte: una secuencia de ocho bits.

En la actualidad se vive en la *era del exabyte* de modo que el usuario ya está acostumbrado a los terabytes que almacenan los discos duros de sus PC o incluso de sus laptops y las empresas ya almacenan en sus bases de datos cantidades de información del orden de petabytes y exabytes.

La **memoria principal** es uno de los componentes más importantes de una computadora y sirve para almacenamiento de información (datos y programas). Existen dos tipos de memoria y de almacenamiento: almacenamiento principal (memoria principal o memoria central) y almacenamiento secundario o almacenamiento masivo (discos, cintas, etcétera).

La memoria central de una computadora es una zona de almacenamiento organizada en centenares o millares de unidades de almacenamiento individual o celdas. La memoria central consta de un conjunto de *celdas de memoria* (estas **celdas** o **posiciones de memoria** se denominan también ***palabras***, aunque no "guardan" analogía con las palabras del lenguaje). Cada palabra puede ser un grupo de 8 bits, 16 bits, 32 bits o incluso 64 bits en las computadoras más modernas y potentes. Si la palabra es de 8 bits se conoce como *byte*. El término *bit* (*dígito binario*) se deriva de las palabras inglesas ***binary digit*** y es la unidad de información más pequeña que puede tratar una compu-

[3] Se adoptó el término **kilo** en computadoras debido a que 1.024 es muy próximo a 1.000, y por eso en términos familiares y para que los cálculos se puedan hacer fáciles mentalmente se asocia 1 KB a 1.000 bytes y 1 MB a 1.000.000 de bytes y 1 GB a 1.000.000.000 de bytes. Así, cuando se habla en jerga diaria de 5 KB estamos hablando, en rigor, de 5 × 1.024 = 5.120 bytes, pero en cálculos consideramos 5.000 bytes. De este modo se guarda correspondencia con las restantes representaciones de las palabras *kilo*, *mega*, *giga*. Usted debe considerar siempre los valores reales para 1 KB, 1 MB o 1 GB mientras esté en su fase de formación y posteriormente en el campo profesional desde el punto de vista de programación, para evitar errores técnicos en el diseño de sus programas, y sólo recurrir a las cifras mil, millón, etc., para la jerga diaria.

Cristian Pilapanta 1095211-2202090019
10952112202090019

tadora (figura 1.3). El término ***byte*** es muy utilizado en la jerga informática y, normalmente, las palabras de 16 bits se suelen conocer como palabras de 2 *bytes*, y las palabras de 32 bits como palabras de 4 *bytes*.

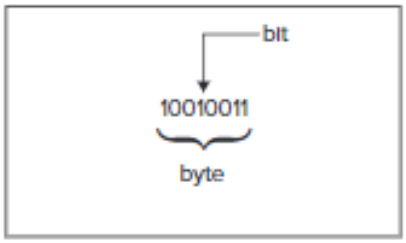


Figura 1.3. Relación entre un bit y un byte.

La mayoría de los equipos actuales tienen al menos 4 u 8 Gb de memoria RAM (en el caso de las *laptops* y PC de escritorio), aunque ya es muy frecuente encontrar computadoras con 10, 12 y 16 Gb e incluso superior. La computadora requiere la mayor capacidad de memoria posible para permitir que el equipo ejecute el mayor número de programas del modo más rápido posible. En caso de que se requiera ampliar la memoria, es posible si se instalan chips o tarjetas de memoria en las ranuras libres que suelen incluir las placas base del sistema. Si la computadora no tiene suficiente memoria, algunos programas se ejecutarán muy lentamente y otros no lo harán de ningún modo.

Los bytes sirven para representar los caracteres (letras, números y signos de puntuación adicionales) en un código estándar internacional denominado **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), utilizado por todas las computadoras del mundo, o bien en un código estándar más moderno denominado **Unicode**. Todos estos símbolos se almacenan en memoria y con ellos trabajan las computadoras

Cristian Pilapanta 1095211-2202090020

1.3.4. Dispositivos de entrada y salida

Para realizar tareas útiles en las computadoras se requiere "tomar" o "capturar" datos y programas, y visualizar los resultados de la manipulación de datos. Los dispositivos (periféricos) que envían (alimentan) datos y programas a las computadoras se denominan dispositivos o periféricos de entrada. Se obtiene información (datos y programas) desde los dispositivos de entrada y ponen esta información a disposición de otras unidades de procesamiento. Los periféricos de entrada más usuales son el teclado y el ratón[4] (*mouse*) y unidades de almacenamiento secundario o escáner para digitalizar información. La información se puede introducir también mediante micrófonos o cámaras de video que permiten "subir" fotografías y videos; o recibir información de redes de comunicaciones, especialmente la red de Internet, o dispositivos de escaneado de texto o imágenes.

Los dispositivos que utiliza la computadora para visualizar y almacenar resultados se denominan **periféricos de salida**. La mayoría de la información que sale de la computadora se visualiza en pantallas, se imprime en papel utilizando impresoras o se almacena en dispositivos de almacenamiento secundario. De igual forma, las computadoras pueden enviar su información a redes como Internet.

En la actualidad son muy populares los periféricos multimedia que se utilizan en tareas multimedia y pueden ser de entrada o de salida: altavoces (suelen estar incorporados en los portátiles, pero también pueden conectarse externamente a través de puertos USB); micrófono (permiten capturar sonidos); webcam (cámara digital web incorporada o conectada externamente también mediante puertos USB; auriculares (permite escuchar sonidos) e incluso auriculares con micrófono incorporado.

Una computadora, normalmente, viene dotada de una gran variedad de puertos de comunicación para cumplir necesidades diversas y atender las necesidades de los periféricos citados anteriormente. Así es común que integren los siguientes puertos (*ports*) de entrada/salida: uno o más puertos de video para conexión de monitores; jacks (co-

[4] Todas las acciones a realizar por el usuario se harán con el ratón con la excepción de las que requieren de la escritura de datos por teclado. El nombre de ratón parece que proviene de la similitud del cable de conexión con la cola de un ratón. Hoy día, sin embargo, este razonamiento carece de sentido ya que existen ratones inalámbricos que no usan cable y se comunican entre sí a través de rayos infrarrojos.

nectores) de audio para conexión de altavoces y(o) micrófonos; puertos USB para conexión de teclados, dispositivos apuntadores (como un ratón), impresoras, cámaras, unidades de disco (duro, discos SSD, memorias *flash*), dispositivos portátiles de almacenamiento (*pendrives* o lápices, memorias SIM), etc. Otros puertos permiten conexión a tarjetas de expansión, como ampliación de memoria, conexión a aparatos de TV, video, audio, etcétera.

1.3.5. Dispositivos de almacenamiento secundario

Los programas y los datos se deben recopilar en la memoria principal antes de su procesamiento, y como esta memoria es volátil toda la información almacenada en ella, antes de su procesamiento, se pierde cuando se apaga la computadora. La información reunida en ella se debe transferir a otros dispositivos de almacenamiento donde se pueda guardar de modo permanente y por largos periodos (con la excepción de que se inutilice o se rompa por alguna causa física o se cambie intencionadamente la información que acumula).

Los datos y los programas se depositan de modo permanente en dispositivos de almacenamiento y se moverán a la memoria cuando la computadora los necesite. Un dispositivo que recopila la información de modo permanente y por largos periodos se denomina almacenamiento secundario o externo. Para poder realizar la transferencia de información desde la memoria principal al almacenamiento secundario o viceversa, estos dispositivos deben ser conectados directamente a la computadora o ellos mismos entre sí. Existen diferentes tipos de dispositivos o periféricos de almacenamiento:

- Unidades de disco (discos duros internos y externos).
- Unidades de discos compactos (CD, DVD).
- Unidades de cinta (ya en desuso).
- Unidades de memoria flash, USB (*pendrives* o lápices) y tarjetas de memoria (SD).
- Unidades de memoria SD (tarjetas de cámaras fotográficas, videocámaras).
- Unidades de disco duro SSD (memorias de estado sólido).
- Unidades de memoria ZIP.

Cristian Palpanta 1095211-2202060021

1095211220206021

1.3.6. Sistemas de comunicación

Las computadoras se conectan entre sí por medio de redes informáticas y dispositivos de comunicación. La conexión física a una red se puede realizar mediante cable, *conexión cableada* redes cableadas y sin cable, redes inalámbricas (*wireless*). Aunque en las empresas existen todavía redes corporativas LAN e Intranet, hoy día lo más común es que tanto las corporativas como las domésticas se conecten entre sí a través de la red Internet y de la computación en la nube.

Los dispositivos de comunicación más utilizados son: módem y tarjeta de interfaz de red (NIC, *network interface card*). En el caso de las redes inalámbricas se requiere un *router* (enrutador, rúter) inalámbrico que permite configurar las redes como si fuesen cableadas.

La conexión de su PC a Internet requerirá la contratación de un proveedor de servicios de Internet (ISP, *Internet Service Provider*) que facilitará la conexión desde el hogar o desde la empresa a Internet. Las conexiones a Internet se realizarán a través de tecnología ADSL (que utiliza la línea telefónica tradicional), cable de fibra óptica, por satélite o mediante tecnologías inalámbricas WiFi o WiMax. Hoy día es muy común encontrar desplegadas redes WiFi en ciudades, aeropuertos, campus de universidades, etc., ya es uno de los medios de comunicación más extendidos.

El otro modelo de conexión a Internet se realiza mediante las *redes móviles (celulares)* y los dispositivos móviles (teléfonos inteligentes y tabletas). Las redes móviles actuales, como se verá posteriormente, vienen soportadas por redes 3G/4G y cada día, más frecuentes, las redes 5G que alcanzan velocidades de conexión muy superiores a las redes cableadas tradicionales y de fibra óptica.

1.4. Software: conceptos básicos y clasificación

Los programas, conocidos como software, son instrucciones a la computadora. Sin programas, una computadora es una máquina vacía. Las computadoras no entienden los lenguajes humanos, de modo que se necesita utilizar lenguajes de computadoras para comunicarse con ellas.

El software de una computadora es un conjunto de instrucciones de programa detalladas que controlan y coordinan los componentes hardware de una computadora así como las operaciones de un sistema informático. El auge de las computadoras en el siglo pasado y en el actual siglo XXI se debe esencialmente al desarrollo de sucesivas generaciones de software potentes y cada vez más amistosas ("fáciles de utilizar").

Las operaciones que debe realizar el hardware son especificadas por una lista de instrucciones, llamadas programas o software. Un **programa de software** es un conjunto de sentencias o instrucciones que se da a la computadora. El proceso de escritura o codificación de un programa se denomina **programación** y las personas que se especializan en esta actividad se denominan **programadores**. Existen dos tipos importantes de software: de sistema y de aplicaciones. Cada tipo realiza una función diferente. Los dos tipos de software están relacionados entre sí, de modo que los usuarios y los programadores pueden hacer así un uso eficiente de la computadora.

El *software de sistema* es un conjunto generalizado de programas que gestiona los recursos de la computadora, como el procesador central, enlaces de comunicaciones y dispositivos periféricos. Las personas que escriben software del sistema se llaman *programadores de sistemas*. El *software de aplicaciones* es el conjunto de programas escritos por empresas o usuarios individuales o en equipo y que instruyen a la computadora para que ejecute una tarea específica. A quienes escriben software de aplicaciones se les conoce como *programadores de aplicaciones*.

Los dos tipos de software están relacionados entre sí, de modo que los usuarios y los programadores pueden hacer un uso eficiente de la computadora. En la figura 1.4 se muestra una vista organizacional de una computadora donde se ven los diferentes tipos de software a modo de capas de la computadora desde su interior (el hardware) hasta su exterior (usuario). Las diferentes capas funcionan gracias a las instrucciones específicas (instrucciones máquina) que forman parte del software de sistema y llegan al software de aplicaciones, desarrollado por los programadores de aplicaciones, que es utilizado por el usuario que no requiere ser un especialista.

El software más importante es el del sistema que incluye el sistema operativo; este software es la colección de programas que permiten utilizar su computadora. Además, está enlazado estrechamente con el hardware del sistema y por consiguiente actúa de interfaz entre el usuario y el hardware de la computadora; si no cuenta con el software del sistema, su computadora no funcionará.

Otro software importante es el de aplicación (tal como programas de correo electrónico o de tratamiento de textos) y el de utilidad (programas diseñados para limpiar su unidad de disco o software diseñado para proteger su computadora mientras navega por Internet). Ambos, tanto el de aplicación como el de utilidad emplean el software del sistema para comunicarse con el hardware.

Cristian Pilapanta 1095211-2202090022

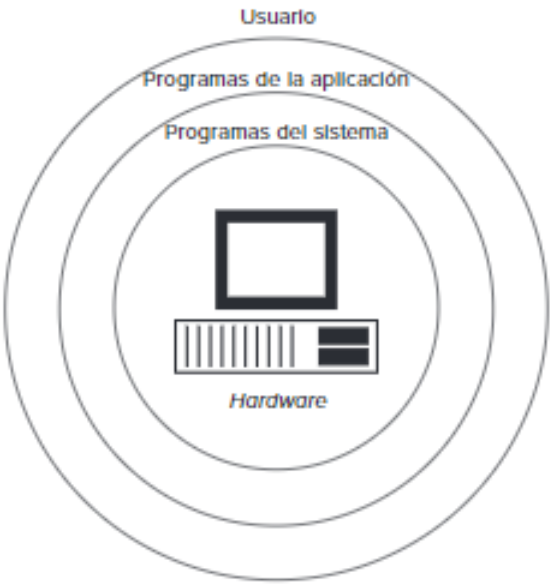


Figura 1.4. Relación entre programas de aplicación y programas de sistema.

1.4.1. Software de sistema

El software de sistema coordina las diferentes partes de un sistema de computadora y conecta e interactúa entre el software de aplicación y el hardware de la computadora. Otro tipo de software de sistema que gestiona, controla las actividades de la computadora y realiza tareas de proceso comunes, se denomina *utility* o **utilidades** (en algunas partes de Latinoamérica se conoce como **utilerías**). El software de sistema que gestiona y controla las actividades de la computadora se denomina *sistema operativo*. Otro software de sistema son los *programas traductores* o de traducción de lenguajes de computadora que convierten los lenguajes de programación, entendibles por los programadores, en lenguaje máquina que entienden las computadoras.

El software de sistema es el conjunto de programas indispensables para que la máquina funcione; se denomina también programas de sistema. Estos programas son, básicamente, el *sistema operativo*, los *editores de texto*, los *compiladores/intérpretes* (lenguajes de programación) y los *programas de utilidad*.

A lo largo de este capítulo y del siguiente se describirán los conceptos fundamentales de los programas de sistemas más utilizados y sobre todo aquellos que afectarán al lector en el aprendizaje de la programación, como sistemas operativos, editores de texto y traductores de lenguajes (compiladores e intérpretes), así como los entornos integrados de desarrollo, herramienta más frecuentemente empleada por los programadores para el desarrollo y ejecución de los programas en los diferentes lenguajes de programación.

1.4.2. Software de aplicaciones

El **software de aplicación** tiene como función principal asistir y ayudar a un usuario de una computadora para ejecutar tareas específicas. Los programas de aplicación se pueden desarrollar con diferentes lenguajes y herramientas de software. Por ejemplo, una aplicación de procesamiento de textos (*word processing*) como Word, Pages o Google Docs, que ayuda a crear documentos; una hoja de cálculo como Lotus 123 o Excel, que ayuda a automatizar tareas tediosas o repetitivas de cálculos matemáticos o estadísticos, a generar diagramas o gráficos; presentaciones visuales como PowerPoint; o a crear bases de datos como Access u Oracle que ayudan a crear archivos y registros de datos.

Los usuarios normalmente compran el software de aplicaciones en discos CD, DVD, unidades pendrive, o en la nube (*cloud*), los descargan e instalan el software copiando los programas correspondientes de los discos en el disco duro de la computadora. Cuando compre estos programas asegúrese de que son compatibles con su computadora y con su sistema operativo. Existe una gran diversidad de programas de aplicación para todo tipo de actividades tanto de modo personal, como de negocios, navegación y manipulación en Internet, gráficos y presentaciones visuales, etcétera.

Los lenguajes de programación sirven para escribir programas que permitan la comunicación usuario/máquina. Unos programas especiales llamados traductores (compiladores o intérpretes) convierten las instrucciones escritas en lenguajes de programación en instrucciones escritas en lenguajes máquina (0 y 1 bits) que ésta pueda entender.

Los programas de utilidad facilitan el uso de la computadora. Un buen ejemplo es un editor de textos que permite la escritura y edición de documentos. Este libro ha sido escrito en un editor de textos o procesador de palabras (*word processor*).

Los programas que realizan tareas concretas, nóminas, contabilidad, análisis estadístico, etcétera, es decir, los programas que podrá escribir, por ejemplo, en C se denominan programas de aplicación. A lo largo del libro se verán pequeños programas de aplicación que muestran los principios de una buena programación de computadora.

Se debe diferenciar entre el acto de crear un programa y la acción de la computadora cuando ejecuta las instrucciones del programa. La creación de un programa se hace inicialmente en papel y a continuación se introduce en la computadora y se convierte en lenguaje entendible por la computadora. La ejecución de un programa requiere una aplicación de una entrada (datos) al programa y la obtención de una salida (resultados). La entrada puede tener una variedad de formas, como números o caracteres alfabéticos. La salida puede también tener formas como datos numéricos o caracteres, señales para controlar equipos o robots, etcétera.

Los programas de utilidad facilitan el uso de la computadora. Un buen ejemplo es un editor de textos que permite la escritura y edición de documentos. Este libro ha sido escrito en un editor de textos o procesador de palabras (*word processor*).

Cristian Palpanta 1095211-2202000023

1.4.3. Software propietario

El software propietario es el de código cerrado, a diferencia del software libre y el de código abierto, su uso, distribución y modificación están limitados y restringidos a los propietarios de su licencia. Este tipo de software no se puede copiar ni modificar por personas o entidades ajenas. Está protegido con leyes de *copyright* y otras leyes de propiedad. Generalmente es de pago y su nivel de calidad, en general, *suele ser* superior al ofrecido por el software de código abierto. Pese a la creciente implantación del código abierto, sigue siendo de uso mayoritario, sobre todo en organizaciones y empresas. El software propietario tiene una licencia específica por la que el cliente paga una cuota o cantidad fija, su código fuente no está abierto de modo universal y su uso es restrictivo. El cliente paga por el producto aceptando condiciones y limitaciones. Los ejemplos de uso propietario son numerosos:

- *Sistema operativo:* Windows de Microsoft y macOS de Apple.
- *Ofimática:* Office de Microsoft, CRM de Salesforce.com, ERP de SAP.
- *Navegadores de Internet:* Microsoft Internet Explorer, Google Chrome, Safari, Mozilla Firefox.
- *Software de antivirus:* Panda, Kaspersky, Norton.
- *Diseño gráfico:* Corel Draw, Photoshop, Adobe Illustrator, Canva.

1.4.4. Software de código abierto

El desarrollo de software de código abierto (*open source*) es un enfoque de desarrollo en el que el código fuente de un sistema de software se publica y se invita a voluntarios a participar en el proceso de desarrollo (Raymond 2001), sus raíces son el software libre de la *Free Software Foundation* (`www.fsf.org`), sus orígenes se remontan a la década de 1980 y a su fundador, Richard Stallman que, a su vez, creó el proyecto GNU y la licencia GPL, que abogaba por que el código fuente no debía ser propietario sino siempre disponible para que los usuarios lo pudieran examinar y modificar como desearan. En su origen, el software de código abierto nació con unos principios de colaboración, asociación, intercambio de información, acceso rápido, menor costo de la tecnología y como evolución del software libre, con ciertas semejanzas pero también bastantes diferencias.

El término "**open source**" (código abierto) fue acuñado por Christine Peterson el 3 de febrero de 1998, en aquel entonces Directora Ejecutiva de Foresight Institute. Su concepto de código abierto implicaba promoción de software seguro, distribuido y desarrollado libremente, de modo que permitiera:

- Posibilidad de descarga del software y pago por los servicios.
- Base del producto gratuita y módulos de pago.
- Modelo *freemium:* una versión gratuita y otra de pago por funcionalidades avanzadas.
- Producto gratuito y actualizaciones de pago.
- Licencias abiertas para negocios "**business source**".

El software abierto es la columna vertebral de Internet y de la ingeniería de software. Linux el sistema operativo más ampliamente utilizado en servidores, así como el servidor web de Apache. Otros productos importantes universalmente utilizados como productos **open-source** son Java, el IDE Eclipse y el sistema de gestión de bases de datos MySQL, así como sistema operativo Android, instalado en millones de dispositivos móviles. Además, la mayoría de los grandes fabricantes de la industria informática como IBM y Oracle soportan el movimiento *open-source* y muchos de sus productos software son de código abierto.

Actualmente, es una tendencia creciente en las empresas y en el caso del software de **Big Data**, las tecnologías fundamentales de **Big Data** y de programación avanzada, utilizan **open-source**, tales como Hadoop, Spark, Hive, Python, etcétera.

Licencias *open-source*

Existe un gran debate sobre la licencia de software (Sommerville, pp. 220-221). Aunque los principios fundamentales del desarrollo de *open-source* establecen que el código fuente debe estar disponible libremente, no significa que

Cristian Pilapanta 1095211-2202090026

el desarrollador pueda hacer lo que crea conveniente con ese código. Legalmente, el desarrollador del código (bien una persona o una empresa) es propietario del código. Existe un gran movimiento y tendencias sobre las licencias de software que puedan ser abiertas, propietarias y que se puedan vender como sistemas de fuente cerrados.

La mayoría de las licencias de *open-source* se certifican o distribuyen en tres modelos:

1. La GNU General Public License (GPL). Esta licencia significa que si se utiliza software de fuente abierta, está protegido y garantiza que los usuarios finales pueden usarlo, estudiarlo, compartirlo o modificarlo; pero siempre será código abierto.
2. La GNU Lesser General Public License (LGPL). Es una variante de la licencia GPL en donde se pueden escribir componentes que enlazan el código fuente sin tener que publicar la fuente de estos componentes. Sin embargo, si se modifica el software con licencia, entonces se debe publicar éste como un *open-source*.
3. El Berkeley Standard Distribution (BS) License. Es una licencia no recíproca, que significa que no se está obligado a reeditar cualquier cambio o modificaciones realizados al código open-source. Se puede incluir el código en sistemas propietarios que son vendidos. Si se utilizan componentes de código abierto se debe agradecer al creador original del código.

La licencia MIT es una variante de la licencia BSD con condiciones similares.

Cristian Pilapanta 1095211-2202090027

1.5. Sistema operativo

Un **sistema operativo** o **SO** (**OS**, *operating system*) es tal vez la parte más importante del software de sistema y es el software que controla y gestiona los recursos de la computadora. En la práctica el sistema operativo es la colección de programas de computadora que controla la interacción del usuario y el *hardware* de la computadora. El sistema operativo es el administrador principal de la computadora, y por ello a veces se le compara con el director de una orquesta, ya que este software es el responsable de dirigir todas las operaciones de la computadora y gestionar todos sus recursos. El núcleo (*kernel*) de un sistema operativo es el software que contiene los componentes fundamentales (*core*) de dicho sistema operativo. El sistema operativo asigna recursos, planifica el uso de recursos y tareas de la computadora, y monitorea las actividades del sistema informático. Estos recursos incluyen memoria, dispositivos de E/S (entrada/salida), y la UCP o CPU (unidad central de proceso). El sistema operativo proporciona servicios como asignar memoria a un programa y manipulación del control de los dispositivos de E/S como el monitor, el teclado o las unidades de disco. La tabla 1.2 muestra algunos de los sistemas operativos más populares utilizados en enseñanza y en informática profesional. Cuando un usuario interactúa con una computadora, la interacción está controlada por el sistema operativo. Un usuario se comunica con un sistema operativo a través de una interfaz de usuario de ese sistema operativo. Los sistemas operativos modernos utilizan una interfaz gráfica de usuario, IGU (*graphical user interface*, GUI) que hace uso masivo de íconos, botones, barras y cuadros de diálogo para realizar tareas que controla el teclado o el ratón, entre otros dispositivos.

El sistema operativo, que sirve, esencialmente, para facilitar la escritura y uso de sus propios programas, dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos), discos y otros dispositivos. Gracias al sistema operativo es posible que el programador pueda introducir y grabar nuevos programas, así como instruir a la computadora para que los ejecute. Los sistemas operativos proporcionan servicios que permiten que cada aplicación se ejecute de modo seguro, eficiente y en concurrencia o en paralelo con otras aplicaciones.

La evolución de los sistemas operativos ha sido considerable desde el nacimiento de las computadoras y han sido numerosos los que han tenido repercusión en las empresas, los negocios, la educación, la investigación, etcétera. Los de mayor impacto han sido desde los primitivos sistemas operativos apoyados por los grandes fabricantes de computación como IBM, Hewlett Packard, Digital, Sun Microsystems, entre otros, hasta los que nacieron a la vez que las computadoras personales, PC, como CP/M (muy popular entre las primeras computadoras personales de 8 bits), MS/DOS (diseñado por Microsoft para las computadoras personales PC), UNIX (el primer sistema operativo abierto), que poco a poco ha ido cediendo su presencia en favor de Linux, el sistema operativo abierto por excelencia en la actualidad y con seguridad en el futuro.

Tabla 1.2. Sistemas operativos: clasificación y características.

Sistema operativo	Características
Windows 10 Windows 8.1	El último sistema operativo de Microsoft es Windows 10 y sus últimas versiones son: 1909 y 1903. La versión Windows 7.1 terminó el soporte el 14 de enero de 2020. Windows 8.1 tiene soporte de Microsoft hasta el 10 de enero de 2023.
Linux	Sistema operativo de software abierto, gratuito y de libre distribución, similar a UNIX, y una gran alternativa a Windows. Muy utilizado en servidores de aplicaciones para Internet.
macOS	Sistema operativo de las computadoras Apple Macintosh. Sus últimas versiones son: macOS Catalina v.10.15.2 y macOS Mojave 10.14.6.
iPhone OS (IOS)	Sistema operativo utilizado por los dispositivos de Apple (iPad, iPod, iPhone). En la actualidad, la última versión de IOS y iPadOS 13.3.
Android	Sistema operativo abierto creado por Google y de gran aceptación en teléfonos inteligentes (*smartphones*) y tabletas (*tablets*). La última versión presentada en septiembre de 2020 es Android 10.
Chrome OS	Sistema operativo de Google específico para Internet.
Windows 10 Mobile	Microsoft anunció en 2019 que la versión 10 Mobile dejará de actualizarse en diciembre de 2019 y recomienda migrar a IOS y Android.
Sistemas operativos de código abierto	FreeBSD (instalado en gran número de servidores; implantado en Netflix), Chrome OS, Haiku, FreeDOS, ReactOS

Cristian Pilapanta 1095211-2202060027

Los sistemas operativos en función del número de usuarios se clasifican en: *monousuarios* (un solo usuario) y *multiusuarios*, o tiempo compartido (diferentes usuarios) y atendiendo al número de las tareas (procesos) que pueden realizar simultáneamente, o hilos de proceso:

- Monotarea (una sola tarea).
- Por lotes (*batch*).
- Multitarea (múltiples tareas).
- Multiprogramación.
- Multiproceso.
- Multihilo.

Desde el punto de vista práctico con enfoque al usuario y a la instalación del sistema existen dos grandes categorías: *sistema operativo para usuario* y *sistema operativo para servidores* (las computadoras que gestionan y controlan las redes de computadoras de las organizaciones y empresas). Los sistemas operativos con mayor penetración en los servidores de las redes de computadoras son: Linux, Windows Server, Mac OS, etcétera. Los servidores son una parte fundamental en los centros de procesos de datos y los programadores e ingenieros de sistemas deberán conocer las características y funcionamiento de los numerosos tipos de servidores existentes como: aplicaciones, audio/video, bases de datos, correo electrónico, impresión, chat, etc., y los dos grandes tipos de servidores desde el punto de vista práctico son los servidores web y los servidores proxy.

Los sistemas operativos son sistemas de software que facilitan el uso más adecuado de las computadoras a los usuarios, desarrolladores de aplicaciones y administradores del sistema. Proporcionan servicios que permiten a cada aplicación ejecutarse con seguridad, eficiencia y de modo concurrente (en paralelo) con otras aplicaciones. El software que contiene los componentes del sistema operativo se llama núcleo (*kernel*). Los sistemas operativos más populares en las computadoras de escritorio y portátiles (laptops) son:

- Linux.
- Windows.
- OS X de Macintosh (antes Mac OS).

Y los sistemas operativos móviles que se utilizan en teléfonos inteligentes y tabletas son:

- Android de Google.
- iOS de Apple (para iPhone, iPad y dispositivos iPod Touch).

Windows de Microsoft

Comercializado desde la década de 1980 es un software propietario administrado, en exclusiva, por Microsoft. Consta de una interfaz gráfica de usuario construida sobre DOS (disk operating system) y muy popular, por ser el sistema operativo más extendido en computadoras de escritorio y portátiles (laptops y notebooks). Se apoya en íconos, menús y ventanas. La última versión del sistema operativo es Windows 10 que incluye el asistente virtual Cortana y un nuevo centro de notificaciones Windows.

Linux

El sistema operativo Linux es el más popular dentro de los sistemas de código abierto. Con este software los usuarios y empresas construyen sus desarrollos y aplicaciones, y contribuyen a su mantenimiento y evolución, facilitando intercambios de uso sin cargo. Muchas empresas multinacionales distribuidoras de software han realizado grandes inversiones y difunden Linux, tales como IBM y Oracle.

El código fuente de Linux está disponible al público para su examen y modificación y es libre y gratuito para descargar e instalar. Los usuarios de Linux se benefician de una gran comunidad de usuarios que activan y depuran continuamente el sistema operativo que se puede personalizar y configurar para cumplir las necesidades que se requieren. Existen numerosas distribuciones de Linux, aunque las más populares son Red Hat Linux y Ubuntu Linux. Con el paso del tiempo, Linux se ha vuelto muy popular en servidores y sistemas embebidos tales como los citados teléfonos inteligentes.

Cristian Pilapanta 1086211-2202060026

Apple

Apple se fundó en 1976; en la actualidad distribuye en exclusiva el sistema operativo macOS para computadoras de escritorio y servidores, e iOS para dispositivos móviles como iPhone, iPad, iPod Touch, Apple Watch y Apple TV. Tiene de soporte el lenguaje Objective-C, aunque desde 2014 introdujo un nuevo lenguaje de programación original de Apple, Swift, que lo está sustituyendo gradualmente y está adquiriendo gran popularidad (ver índice Tiobe, enero 2020).

Android de Google

Es el sistema operativo más utilizado en dispositivos móviles. Se basa en el sistema operativo Linux y en Java, aunque las aplicaciones de Android también se pueden desarrollar en C y C++. El sistema operativo es de código abierto y gratuito. Fue desarrollado por la empresa Android Inc., que fue adquirida por Google en 2005. En 2007 se creó la organización *Open Handset Alliance*,[5] cuyo objetivo es desarrollar, mantener y evolucionar Android, pensando en innovación de tecnologías móviles, mejora y desarrollo de experiencia de usuario y con costos reducidos.

Sistemas operativos móviles: aplicaciones móviles

Las aplicaciones (programas de aplicación) para móviles (celulares) son las creadas de manera expresa para entornos o plataformas móviles, fundamentalmente teléfonos inteligentes y tabletas. Existen dos tipos de aplicaciones móviles: nativas (*apps*) y web.

- **Aplicación móvil nativa** (*app*). Son aplicaciones diseñadas para ejecutarse sobre una plataforma móvil y un sistema operativo concreto (iOS y Android tienen la mayor cuota de mercado en la actualidad). Necesitan descargarse de una tienda de aplicaciones (App Store de Apple, Google Play de Android o de otras empresas como Telefónica, Vodafone, Verizon o Telmex). Se descargan, instalan y ejecutan en el dispositivo móvil de modo gratuito o con el pago de una cuota.

[5] www.openhandsetialliance.com

- **Aplicación web móvil.** Se desarrollan con tecnologías web y se puede acceder a ellas y visualizarlas con un navegador de Internet (Safari, Firefox, Chrome, Opera, Internet Explorer).

Los lenguajes de programación empleados en el desarrollo de aplicaciones móviles son muy variados y van desde los lenguajes tradicionales como C++, Java, Python, HTML5 hasta clásicos de la programación orientada a objetos como Objective-C y el nuevo Swift utilizados por Apple para sus desarrollos.

1.6. El lenguaje de la computadora

Los humanos nos entendemos con los lenguajes naturales como el español, el inglés o el portugués, pero, ¿y las computadoras cómo se entienden? Cuando pulsamos la letra B en el teclado se visualiza la letra B en la pantalla del monitor. ¿Cuál es la información que se ha almacenado en la memoria central de la computadora? En resumen, ¿cuál es el lenguaje que entienden las computadoras y cómo se almacena cualquier otra información que se introduce en el teclado o se lee de una memoria USB?

Recordemos que una computadora es un dispositivo electrónico que procesa información de modo automático. Las señales eléctricas se mueven entre los diferentes componentes de una computadora. Existen dos tipos de señales eléctricas: analógica y digital. Las señales analógicas se utilizan para representar cosas como sonido; las cintas de audio, por ejemplo, almacenan los datos en señales analógicas.

Las computadoras funcionan con señales digitales, las cuales representan la información como una secuencia de ceros y unos. Un 0 representa una señal de bajo voltaje, y un 1, una señal de alto voltaje. Un tema vital en el proceso de funcionamiento de una computadora es estudiar la forma de representación de la información en dicha computadora. Es necesario considerar cómo se puede codificar la información en patrones de bits que sean fácilmente almacenables y procesables por los elementos internos de la computadora. Las señales digitales se procesan en el interior de las computadoras y por esta razón, el lenguaje de una computadora, llamado lenguaje máquina, es una secuencia de dígitos 0 y 1 (*bits*). Normalmente una secuencia de ceros (0) y unos (1) se conoce como código o números binarios.

Cristian Pilapanta 1095211-2202080029

El sistema de numeración que se utiliza en la vida diaria se denomina código decimal o de base 10. En una computadora los caracteres se representan como secuencias de dígitos 0 y 1, es decir, números binarios; el sistema de numeración que utiliza una computadora se llama código binario o de base 2.

1.6.1. Representación de la información en las computadoras (códigos de caracteres)

Cuando se pulsa un carácter: una letra, un número o un símbolo especial (como &, %) en su teclado, se almacena en la memoria principal de una computadora una secuencia de bits que es diferente en cada caso. El código o esquema más utilizado es el código **ASCII** (*American Standard Code for Information Interchange*). El código ASCII (apéndice B) consta de 128 caracteres (27 = 128, 0 a 127), de modo que la posición del primer carácter es 0, la posición del segundo carácter es 1 y así sucesivamente. Por ejemplo, el carácter A es el carácter 66 y de posición 65; el carácter B es el carácter 67 y de posición 66 (tabla 1.3). En el interior de las computadoras cada carácter se representa por una secuencia de ocho bits, es decir un byte. La representación binaria del código A es 01000001 y el carácter 4 es 00110100.

Otro conjunto de caracteres muy utilizado en computadoras es el código **Unicode** que consta de 65.536 caracteres, en lugar de los 128 caracteres del código ASCII básico. Java utiliza el código Unicode en el que cada carácter se representa por 2 bytes, 16 bits, en lugar de 1 byte en el código ASCII. La gran ventaja de Unicode es que permite representar caracteres de numerosos idiomas internacionales como el chino, indio, ruso, etcétera.

Tabla 1.3. Códigos de representación de la información.

Carácter	Código ASCII	Código binario
A	65	01000001
B	66	01000010
C	67	01000011
—	—	—
1	49	00110001
2	50	00110010
3	51	00110011
—	—	—

1.7. La programación de las computadoras en perspectiva

Las computadoras, en la actualidad, se utilizan para innumerables aplicaciones: predecir el tiempo, diseñar barcos y aviones, realizar películas, gestionar negocios, realizar transacciones financieras o controlar fábricas. Una computadora moderna podría definirse como "una máquina que almacena y manipula información bajo el control de un programa cambiante y actualizable". Se deduce de esta definición que ponemos información en la computadora y se puede transformar la información en notas, formatos útiles y, a continuación, extraer o visualizar la información para su interpretación.

Un programa de computadora es un conjunto detallado de instrucciones paso a paso que indican al dispositivo las acciones a realizar con exactitud. Si cambiamos el programa, entonces la computadora realiza una secuencia diferente de acciones y, por consiguiente, ejecutará una tarea diferente. Esta característica facilita que la PC pueda actuar en un momento como un procesador de texto, en el siguiente momento como un planificador financiero y luego como una máquina de juegos. La máquina permanece igual pero el programa que controla la máquina cambia. En esencia, cada computadora es una máquina para ejecutar un programa.

El software (los programas) regulan el hardware (las máquinas físicas). El software es quien determina lo que puede hacer cualquier computadora. Sin software, las computadoras sólo serían una máquina de adorno o "una máquina tonta". El proceso de creación de software se llama **programación**, eje central de este libro.

La programación de computadoras es una actividad interesante y requiere la capacidad y actitud para ver, por ejemplo, grandes dibujos y prestar atención a los detalles. La programación es una parte fundamental de la ciencia de las computadoras (informática o ingeniería de sistemas) y, por consiguiente, muy atractiva e importante para cualquier persona interesada en convertirse en un profesional de computadoras. La comprensión de las fortalezas y limitaciones de las herramientas de computación requiere una comprensión de la programación. Así, hoy día, los programadores tienen una gran demanda, pero les exige un gran aprendizaje y formación continua.

Para poder realizar procesos que funcionen en las computadoras, los programadores deben utilizar numerosas técnicas de investigación, siendo las tres principales: análisis, diseño y evaluación o experimentación.

Un medio para demostrar que un problema específico se puede resolver es diseñar una solución. Es decir, desarrollar un proceso paso a paso (*step-to-step*) para conseguir el resultado deseado. Este proceso se denomina **algoritmo**. El diseño de algoritmos es una de las actividades más importantes de la informática y la ingeniería de sistemas, y por extensión a las nuevas disciplinas de ciencia de datos y las técnicas avanzadas de inteligencia artificial como aprendizaje automático (*machine learning*) y aprendizaje profundo (*deep learning*). El análisis es el proceso de examinar algoritmos y problemas de modo matemático. El **análisis de algoritmos** es una parte importante de la ciencia de las computadoras y uno de los objetivos centrales de este libro es mostrar cómo afrontar el análisis y su diseño posterior correspondiente, que luego se convertirá en un programa que ejecutará la computadora.

Sin embargo, en el paraguas de computación, caen un gran número de actividades o disciplinas: computación móvil, *networking* (redes de comunicaciones), interacción persona-computadora, inteligencia artificial, ciencia computacional, bases de datos, Big Data, minería de datos, ingeniería de software, diseño web, diseño multimedia, producción musical, administración de sistemas de información y cada vez con mayor énfasis, seguridad de las computadoras y la ciberseguridad.

Cristian Pilapanta 1095211-2202090038

1.8. Lenguajes de programación

Recordemos que un programa es sólo una secuencia de instrucciones que indican a una computadora las tareas a realizar. Evidentemente se necesita proporcionar estas instrucciones en un lenguaje que la computadora pueda comprender. Aunque las computadoras reconocen cada vez más nuestra voz con tecnologías como Siri (Apple), Google Assistance o Now (Android), Alexa (Amazon) o Cortana (Microsoft), es necesario diseñar un programa de computadora que comprenda totalmente el lenguaje humano y, por ahora, estas actividades son todavía limitadas. El lenguaje natural sigue siendo difícil de entender por las computadoras, aunque técnicas de inteligencia artificial como reconocimiento de lenguajes, redes neuronales, entre otras, avanzan con gran rapidez.

Las notaciones especiales para la comprensión de las computadoras se llaman *lenguajes de programación*, cada estructura de un lenguaje de programación tiene un formato preciso (*su sintaxis*) y un significado también preciso (*su semántica*). Un lenguaje de programación es similar a un código de escritura que una computadora seguirá; de hecho, los programas se llaman también *código de la computadora* y el proceso de escribir un algoritmo en un lenguaje de programación se llama *codificación*.

Existen numerosos lenguajes de programación, se han desarrollado literalmente cientos como Python, Java, C, C++, C#, JavaScript, Ruby, Swift, Visual Basic .NET o los tradicionales Basic, Cobol y Fortran y Pascal. Aunque difieren en muchos detalles, todos comparten la propiedad de tener sintaxis y semántica bien definida y no ambigua. Todos los lenguajes anteriores se denominan *lenguajes de alto nivel*. Son precisos y están diseñados para ser comprendidos por los humanos, pero hablando estrictamente, el hardware de la computadora sólo comprende un *lenguaje de bajo nivel*, conocido como *lenguaje máquina*.

Las instrucciones de lenguajes de alto nivel como `c = a + b` o `print "hola"` han de ser traducidas del lenguaje de alto nivel a un lenguaje máquina que la computadora pueda ejecutar. Existen dos formas de hacerlo: puede ser *compilado* o *interpretado*. Como se ha visto en el apartado anterior, para que un procesador realice un proceso se le debe suministrar en primer lugar un algoritmo adecuado. El procesador debe ser capaz de interpretar el algoritmo, lo que significa:

- Comprender las instrucciones de cada paso.
- Realizar las operaciones correspondientes.

Cuando el procesador es una computadora, el algoritmo se debe expresar en un formato que se denomina **programa**, ya que el pseudocódigo o el diagrama de flujo no son comprensibles para la computadora, aunque pueda entenderlos cualquier programador. Un programa se escribe en un lenguaje de programación y las operaciones que conducen a expresar un algoritmo en forma de programa se llaman **programación**. Así pues, para escribir programas de computadoras se utilizan los lenguajes de programación, y **programadores** son los escritores y diseñadores del programa. El proceso de traducir un algoritmo en pseudocódigo a un lenguaje de programación se denomina **codificación**, y el algoritmo escrito en un lenguaje de programación se denomina **código fuente**.

En la realidad, la computadora no entiende directamente los lenguajes de programación, sino que se requiere un programa que traduzca el código fuente a otro lenguaje que sí entiende pero es muy complejo para las personas; este lenguaje se conoce como **lenguaje máquina** y el código correspondiente **código máquina**. Los programas que traducen el código fuente escrito en un lenguaje de programación, tal como C++, a código máquina se denominan **traductores**. El proceso de conversión de un algoritmo escrito en **pseudocódigo** hasta un programa ejecutable comprensible por la máquina se muestra en la figura 1.5.

Hoy día, la mayoría de los programadores emplea lenguajes de programación como los mencionados anteriormente. Estos lenguajes se denominan lenguajes de alto nivel y permiten a los profesionales resolver problemas convirtiendo sus algoritmos en programas escritos en alguno de estos lenguajes de programación.

Los **lenguajes de programación** se utilizan para escribir programas. Los programas de las computadoras modernas constan de secuencias de instrucciones que se codifican como secuencias de dígitos numéricos que podrán entender dichas computadoras. El sistema de codificación se conoce como lenguaje máquina, que es el lenguaje nativo de una computadora. Desgraciadamente la escritura de programas en lenguaje máquina es una tarea tediosa y difícil ya que sus instrucciones son secuencias de 0 y 1 (patrones de bit, como 11110000, 01110011), que son muy difíciles de recordar y manipular por las personas. En consecuencia, se necesitan lenguajes de programación "amigables con el programador" que permitan escribir los programas para poder *charlar* con facilidad con las computadoras. Sin embargo, las computadoras sólo entienden las instrucciones en lenguaje máquina, por lo que será preciso traducir los programas resultantes a lenguaje máquina antes de poder ejecutarse.

Cristian Palapunta 1095211-2200050039

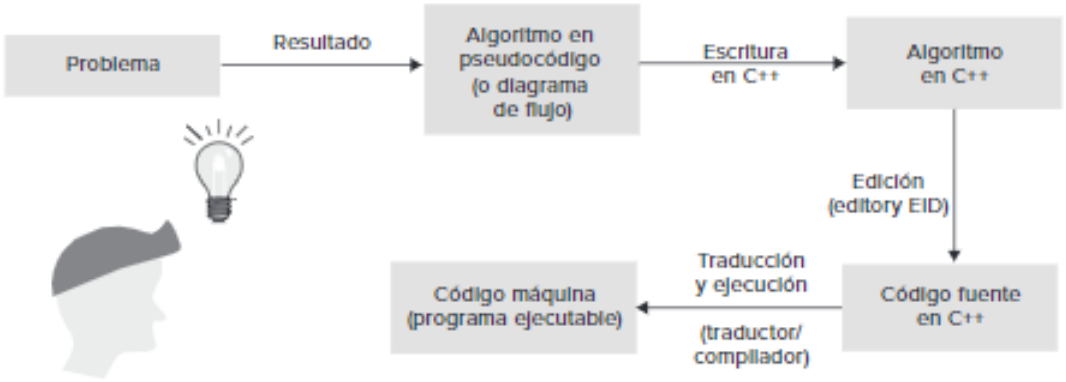


Figura 1.5. Proceso de transformación de un algoritmo en pseudocódigo a un programa ejecutable.

Cada lenguaje de programación tiene un conjunto o "juego" de instrucciones (acciones u operaciones que debe realizar la máquina) que la computadora podrá entender directamente en su código máquina o bien se traducirán a dicho código máquina. Las instrucciones básicas y comunes en casi todos los lenguajes de programación son:

- *Instrucciones de entrada/salida*. Instrucciones de transferencia de información entre dispositivos periféricos y la memoria central, como "leer de..." o bien "escribir en...".
- *Instrucciones de cálculo*. Instrucciones para que la computadora pueda realizar operaciones aritméticas.
- *Instrucciones de control*. Instrucciones que modifican la secuencia de la ejecución del programa.

Cristian Pilapanta 1095211-2202090040

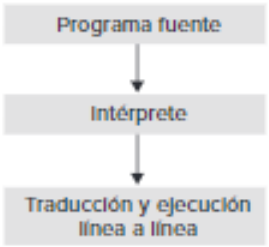


Figura 1.6. Intérprete.

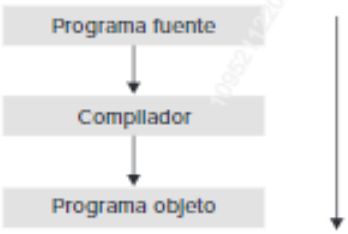


Figura 1.7. La compilación de programas.

Además de estas instrucciones y dependiendo del procesador y del lenguaje de programación, existirán otras que conformarán el conjunto de instrucciones y, junto con las reglas de sintaxis permitirán escribir los programas de las computadoras.

1.8.1. Tipos de lenguajes de programación: máquina, ensamblador y de alto nivel

Los programadores escriben las instrucciones en diferentes lenguajes de programación, algunos comprensibles directamente por las computadoras y otros que requieren etapas de traducción del lenguaje específico a lenguaje de la máquina para que sea comprensible por ésta. Así, se pueden considerar tres grandes tipos de lenguajes de programación: **máquina**, **ensamblador** y **alto nivel**.

Lenguaje máquina

Las computadoras comprenden directamente sólo su propio lenguaje, conocido como **lenguaje máquina** (también llamado **código máquina)** y definido por su arquitectura hardware. Los lenguajes de máquina normalmente constan de dígitos binarios (dígitos 0 y 1) y suelen ser incomprensibles para los humanos debido a su complejidad.

Una computadora puede comprender directamente instrucciones escritas sólo en su propio lenguaje máquina y que está definido por su diseño *hardware*. Los lenguajes máquina constan, generalmente, de cadenas de números binarios (dígitos 0 y 1) que instruyen a las computadoras a realizar operaciones elementales una a una. Los lenguajes máquina son dependientes de la máquina (un lenguaje máquina específico sólo se puede utilizar en una única computadora). Tales lenguajes son incomprensibles para los humanos.

La mayoría de los lenguajes básicos de computadora, como el **lenguaje máquina**, proporcionan las instrucciones a la computadora en bits. Aunque la mayoría de las computadoras utilizan los mismos tipos de operaciones, los diseñadores de los diferentes procesadores (CPU) normalmente eligen conjuntos distintos de códigos binarios para realizar dichas operaciones. Por consiguiente, el lenguaje máquina de una computadora no es necesariamente el mismo que el lenguaje máquina de otra computadora. La única coincidencia entre computadoras es que en cualquiera de ellas, todos los datos se almacenan y manipulan en código binario.

Las primeras computadoras eran programadas en lenguaje máquina (código binario). La programación escrita que utiliza el lenguaje máquina es un proceso tedioso y difícil de realizar. Por ejemplo, para sumar dos números se puede requerir una instrucción como ésta:

1101101010101001

Para realizar cualquier operación, como la anterior, el programador tiene que recordar las posiciones de los datos en memoria. Esta necesidad de recordar los códigos específicos en binario hace muy difícil la programación en código máquina y muy propensa a errores.

Cristian Pilapanta 1005211-2202000041

Lenguaje ensamblador

La programación en lenguaje máquina es muy lenta y tediosa para la mayoría de los programadores. Por ello, para hacer la tarea del programador más fácil comenzaron a crearse lenguajes denominados ensambladores, los cuales, en lugar de utilizar series de números o cadenas que entiendan las computadoras, utilizan palabras, abreviaturas en inglés que pueden entender los humanos y al mismo tiempo representan las operaciones elementales. Dichas abreviaturas constituyeron el fundamento del lenguaje ensamblador. Sin embargo, aunque es mucho más fácil escribir instrucciones en lenguaje ensamblador, una computadora no puede ejecutar instrucciones directamente en este lenguaje debido a que sólo entiende lenguaje máquina. El surgimiento de lenguajes de programación más parecidos al lenguaje humano requirió a su vez el desarrollo de programas traductores que convirtieran instrucciones escritas en lenguaje ensamblador (*assembly language*) a instrucciones en lenguaje máquina. A estos programas traductores se les llama **ensambladores.**

En lenguaje ensamblador, una instrucción tiene un formato más fácil de recordar, llamado nemotécnico o nemónico. Las instrucciones en lenguaje ensamblador son abreviaturas o segmentos de palabras clásicas en inglés. Por ejemplo, algunas instrucciones se recogen en la tabla 1.4.

Tabla 1.4. Ejemplos de instrucciones en lenguaje ensamblador.

Lenguaje ensamblador	Lenguaje máquina	Significado
Add	00100100	Suma
Sub	00100010	Resta
Mult	00100110	Multiplicar
Sto	00100010	Almacenar

Ensamblador

Un programa que traduce un programa escrito en lenguaje ensamblador en un programa equivalente en lenguaje máquina.

El lenguaje ensamblador hizo la programación más fácil, sin embargo, el programador, para conseguir potencia y eficiencia en los programas, tenía que seguir pensando en términos de instrucciones de máquina con la dificultad que eso lleva consigo.

Lenguajes de alto nivel

Con el advenimiento de los lenguajes ensambladores creció el uso de las computadoras, pero los programadores tenían que utilizar numerosas instrucciones, incluso para las tareas más sencillas.

Para poder aumentar la velocidad y facilidad del proceso de programación, se desarrollaron los lenguajes de alto nivel en los que se crearon sentencias o instrucciones que se podían escribir para realizar tareas complejas. Estos lenguajes de alto nivel estaban más próximos a los lenguajes hablados como el inglés, francés o español y además podían contener notaciones matemáticas. Así aparecieron lenguajes como Cobol, Basic, Fortran, Pascal, C, C++, Java o C#, denominados lenguajes de alto nivel y cuyas instrucciones eran: `print`, `read`, `open`, `write`, `for`, `while`, etcétera. Las instrucciones son similares a palabras ordinarias del lenguaje inglés y también contienen notaciones matemáticas relevantes.

Desde el punto de vista de programación, los lenguajes de alto nivel son preferibles a los lenguajes ensamblador y de máquina.

Al igual que sucede con los lenguajes ensambladores las computadoras no pueden ejecutar directamente instrucciones escritas en un lenguaje de alto nivel. Se necesita un *programa traductor* denominado **compilador** o **intérprete** que traduzca las instrucciones en lenguaje de alto nivel a instrucciones en lenguaje máquina, bien directamente como en el caso de C++ o C, o bien un lenguaje intermedio llamado *bytecode* que luego se interpreta en un lenguaje máquina en el caso de Java. El proceso de traducción se denomina **compilación** o **interpretación** de un programa y es un proceso lento como sucede con la traducción de un idioma a otro, aunque las herramientas actuales realizan los procesos de traducción de un modo rápido y eficiente. Los lenguajes de programación pueden ser **compilados** o **interpretados**, según el tipo de traductor que emplean (**compilador** o **intérprete**). Posteriormente se describirán con más detalles los programas traductores, compiladores e intérpretes.

Cristian Pilapanta 1095211-2202090043

El proceso de programación

La programación es un proceso de resolución de problemas. Para resolver estos problemas se requieren técnicas diferentes que comprenden desde el análisis del problema, especificación de requisitos o requerimientos y las etapas de diseño denominadas algoritmos.

Un **algoritmo** es el concepto fundamental de la ciencia de las computadoras (informática). Desde un punto de vista práctico, un algoritmo es un conjunto de pasos que definen cómo se realiza una tarea. Por ejemplo, hay un algoritmo para construir un modelo de avión, de un tren, una lavadora o un aparato de televisión. Antes de que una máquina pueda ejecutar una tarea, se debe diseñar un algoritmo que posteriormente se pueda convertir en un programa que entienda la computadora, para lo cual se necesitará traducir el programa en lenguaje máquina mediante un traductor (compilador o intérprete).

Algoritmo

Proceso de resolución de problemas compuesto por un conjunto de instrucciones que se realizan paso a paso para conseguir una solución y que se obtiene en un tiempo finito.

El estudio de algoritmos es uno de los temas centrales en estudios de ciencias e ingeniería y en particular en ingeniería de sistemas computacionales y en ingeniería informática. El conocimiento de las técnicas de diseño de algoritmos es fundamental para el programador y uno de los temas centrales de nuestra obra.

Para desarrollar un programa que pueda resolver un problema, se debe comenzar analizando y examinando el problema con detenimiento con objeto de obtener diferentes opciones para encontrar la solución; a continuación se selecciona una opción y se debe diseñar el algoritmo correspondiente escribiendo las instrucciones del programa en un lenguaje de alto nivel, etapa conocida como *codificación del programa*, y a continuación introducir y ejecutar el programa en la computadora.

Las computadoras no entienden los algoritmos, por lo que es necesario indicarles exactamente las "acciones que deben hacer" en un lenguaje comprensible para la máquina. La descripción de las acciones o tareas que debe hacer la computadora se denomina "**programa**", y **programación** a la actividad de escribir y verificar tales programas. Por esta razón, se necesitará convertir el algoritmo en un programa mediante un lenguaje de programación cuyas instrucciones sí entienda la computadora. El proceso de resolución de problemas (proceso de programación) con una computadora consta de las siguientes etapas:

- Análisis del problema, con estudio de los requisitos y del dominio del problema.
- Diseño del algoritmo que resuelva el problema.
- Verificar el funcionamiento correcto del algoritmo.
- Implementar el algoritmo en un lenguaje de programación como Java.
- Ejecutar el programa.
- Depurar el programa para obtener y corregir errores.
- Mantener el programa.
- Actualizar el programa.

En el capítulo 2 profundizaremos en el proceso de programación que nos permita adquirir las competencias necesarias para iniciar el aprendizaje de las técnicas de programación y obtener los conocimientos precisos a fin de comenzar a diseñar y construir programas.

Cristian Pilapanta 1085211-2202080044

1.9. Traductores de lenguaje: el proceso de traducción de un programa

El proceso de traducción de un programa fuente escrito en un lenguaje de alto nivel a un lenguaje máquina comprensible por la computadora se realiza mediante programas llamados "**traductores**". Los traductores de lenguaje son programas que traducen los programas fuente escritos en lenguajes de alto nivel a código máquina y se clasifican en dos grandes grupos: **compiladores** e **intérpretes**.

Una de las grandes ventajas de los lenguajes de alto nivel es la portabilidad del proceso de traducción. El lenguaje máquina de una computadora se crea por los diseñadores para una CPU o procesador específico. Cada tipo de computadora tiene su propio lenguaje máquina. Un programa para un procesador i7 de Intel en su laptop no funciona directamente en una CPU ARMv8 en su teléfono inteligente. En resumen, un programa escrito en un lenguaje de alto nivel se puede ejecutar en diversos tipos de computadoras siempre que haya un compilador o intérprete adecuado (que, en esencia, es otro programa). Como resultado, se puede ejecutar el mismo programa escrito en Python en una laptop y en una tableta inteligente incluso aunque tengan procesadores diferentes, ya que ambos soportan un intérprete Python.

1.9.1. Intérpretes

Un *intérprete* es un programa traductor de un lenguaje de alto nivel que traduce y ejecuta directamente las instrucciones escritas en dicho lenguaje de programación. Un intérprete es un traductor que toma un programa fuente, lo traduce y, a continuación, lo ejecuta. Los programas intérpretes actuales más utilizados son JavaScript, Python y Ruby. Los lenguajes intérpretes históricos son: Basic, que ya prácticamente no se utiliza aunque Microsoft comercializa todavía la versión Visual Basic (VB .NET), y Smalltalk, lenguaje de programación orientado a objetos puro.

El programa intérprete traduce el programa escrito en el lenguaje de alto nivel, lo analiza y ejecuta directamente, instrucción a instrucción a medida que sea necesario y sin generar ningún código equivalente. Su acción equivale a la de un intérprete humano, que traduce las frases que oye sobre la marcha, sin producir ningún escrito permanente. En la figura 1.8 se muestra el proceso.

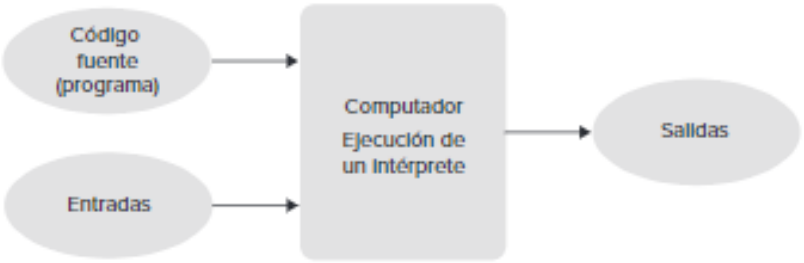


Figura 1.8. Interpretación de un lenguaje de alto nivel intérprete.

1.9.2. Compiladores

Un *compilador* es un programa que traduce los programas fuente escritos en lenguaje de alto nivel a lenguaje máquina. La traducción del programa completo se realiza en una sola operación denominada compilación del programa; es decir, se traducen todas las instrucciones del programa en un solo bloque. El programa compilado y depurado (eliminados los errores del código fuente) se denomina programa ejecutable porque ya se puede ejecutar directamente y cuantas veces se desee; sólo deberá volver a compilarse de nuevo en el caso de que se modifique alguna instrucción del programa. De este modo, el programa ejecutable no necesita del compilador para su ejecución. Los lenguajes compilados típicos más utilizados son: **C**, **C++**, **Java**, **C♯**, **Pascal**, **Fortran** y **Cobol**.

El proceso de traducción (figura 1.9) se denomina **compilación**. El programa de alto nivel se llama código fuente y el código máquina resultante es un programa que se puede ejecutar directamente en la computadora. La línea punteada en el diagrama representa la ejecución del código máquina (ejecución del programa).

Figura 1.9. Compilación de un lenguaje de alto nivel.

1.9.3. Compiladores *versus* Intérpretes

La diferencia entre interpretación y compilación es que esta última es un proceso de traducción de una sola vez. Cuando un programa es compilado se puede ejecutar una y otra vez sin necesidad de utilizar de nuevo el compilador o código fuente. En el caso de un programa interpretado, el intérprete y el código se necesitan cada vez que el programa se ejecuta. Los programas compilados tienden a ser más rápidos, ya que la traducción se hace sólo una vez y completa; sin embargo, los lenguajes interpretados se conducen a sí mismos a un entorno de programación más flexible ya que los programas pueden ser desarrollados y ejecutados interactivamente.

Cristian Pilapanta 1095211-2202090044

La principal diferencia entre compilador e intérprete es que un lenguaje compilado requiere un paso adicional antes de ser ejecutado, la *compilación*, que convierte el código fuente a módulo máquina, mientras que un lenguaje interpretado se convierte a lenguaje máquina a medida que se ejecuta.

Tanto compiladores como intérpretes son programas que convierten el código fuente a lenguaje máquina (instrucciones que entiende la computadora [procesador] en código binario). En los compiladores es necesario realizar el proceso de compilación cada vez que se cambia el código fuente, aunque existen herramientas que pueden automatizar el proceso. Otra desventaja de un lenguaje compilado es que cuando se compila un programa, se deben crear ejecutables para cada uno de los sistemas operativos a utilizar. Un ejecutable creado para el sistema operativo Linux no sirve para Windows.

Sin embargo, un lenguaje compilado es mucho más rápido que uno interpretado, ya que cuando el programa es ejecutado ya está en código máquina y esta circunstancia optimiza el rendimiento. Además, la velocidad es otra desventaja del lenguaje interpretado ya que necesita tener instalado el programa intérprete, condición que no es necesaria en un lenguaje compilado que se convierte directamente a lenguaje máquina.

Existen tres modos de implementación de lenguajes:

- Un lenguaje puede ser interpretado.
- Un lenguaje puede ser compilado a un lenguaje máquina.
- Un lenguaje puede ser implementado a alguna combinación de los métodos anteriores.

El programa compilado es más fácil de distribuir, pero sólo se puede ejecutar en la plataforma que se ha compilado. Cada sistema operativo (y procesador) necesita una versión compilada diferente. C++ se podría ejecutar en una computadora con un procesador x86 pero no en uno con ARM, o bien en Linux pero no en Windows. Un programa compilado es más seguro que uno interpretado, porque no contiene el código fuente que puede ser modificado incorrectamente por el usuario.

La programación en un lenguaje compilado es un proceso de tres etapas:

- Escribir el programa fuente.
- Compilar el programa fuente produciendo un programa ejecutable.
- Se debe ejecutar el programa ejecutable.

Programación en un lenguaje interpretado, es un proceso de dos pasos:

- Primero, escribir el programa fuente.
- A continuación, se ejecuta el programa mediante la ejecución del intérprete.

1.9.4. La compilación y sus fases

La *compilación* es el proceso de traducción de programas fuente a programas objeto. El programa objeto obtenido de la compilación ha sido traducido normalmente a código máquina.

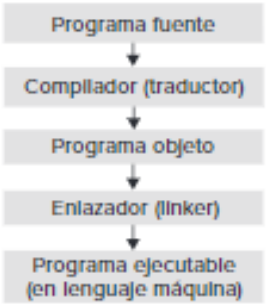


Figura 1.10. Fases de la compilación.

Cristian Pilapanta 1095211-2200060045

Para conseguir el programa máquina real se debe utilizar un programa llamado *montador* o *enlazador* (*linker*). El proceso de montaje conduce a un programa en lenguaje máquina directamente ejecutable (figura 1.10).

El proceso de ejecución de un programa escrito en un lenguaje de programación y mediante un compilador suele tener los siguientes pasos:

1. Escritura del *programa fuente* con un *editor* (programa que permite a una computadora actuar de modo similar a una máquina de escribir electrónica) y guardarlo en un dispositivo de almacenamiento (por ejemplo, un disco).
2. Introducir el programa fuente en la memoria.
3. *Compilar* el programa con el compilador del lenguaje de programación.
4. *Verificar* y *corregir errores de compilación* (listado de errores).
5. Entrega del programa *objeto*.
6. El enlazador obtiene el *programa ejecutable*.
7. Se ejecuta el programa y, si no existen errores, se tendrá la salida del programa. El proceso de ejecución se muestra en las figuras 1.11 y 1.12.

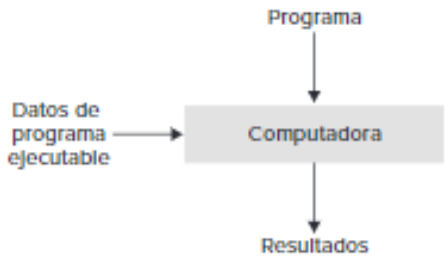


Figura 1.11. Ejecución de un programa.

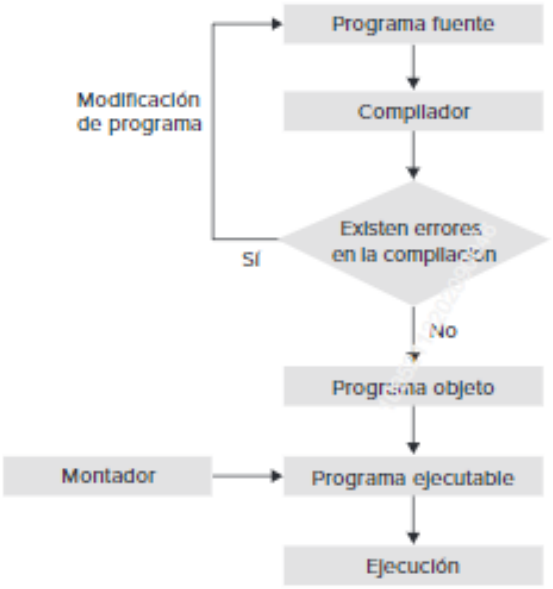


Figura 1.12. Fases de ejecución de un programa.

1.10. Evolución de los lenguajes de programación

En la década de 1940 cuando nacían las primeras computadoras digitales, el lenguaje que se utilizaba para programar era el *lenguaje máquina*, el cual traducía directamente el código máquina (código binario) comprensible para las computadoras. Las instrucciones en lenguaje máquina dependían de cada computadora y debido a la dificultad de su escritura, los investigadores de la época simplificaron el proceso de programación inventando sistemas de notación en los cuales las instrucciones se representaban en formatos *nemónicos* (**nemotécnicos**) en vez de en formatos numéricos, que eran más difíciles de recordar. Por ejemplo, mientras la instrucción:

```
Mover el contenido del registro 4 al registro 8
```

se podía expresar en lenguaje máquina como

```
4048  o bien  0010 0000 0010 1000
```

en código nemotécnico podía aparecer como

```
MOV  R5,  R6
```

Cristian Pilapanta 1095211-2202090046

Para convertir los programas escritos en código nemotécnico a lenguaje máquina, se crearon programas ensambladores. Es decir, los ensambladores son programas que traducen programas escritos en código nemotécnico en instrucciones numéricas en lenguaje máquina que son compatibles y legibles por la máquina. Estos programas de traducción se llaman ensambladores, porque su tarea es ensamblar las instrucciones reales de la máquina con los nemotécnicos e identificadores que representan las instrucciones escritas en ensamblador. A estos lenguajes se les denominó de segunda generación, reservando el nombre de primera generación para los lenguajes de máquina.

En las décadas de 1950 y 1960 comenzaron a desarrollarse lenguajes de programación de tercera generación que diferían de las generaciones anteriores en que sus instrucciones o primitivas eran de alto nivel (comprensibles por el programador, como si fueran **lenguajes naturales**) e **independientes de la máquina**. Estos lenguajes se llamaron lenguajes de alto nivel. Los ejemplos más conocidos son **Fortran** (FORmula TRANslator), que fue desarrollado para aplicaciones científicas y de ingeniería, y **Cobol** (COmmon Business Oriented Language), que fue inventado por la U.S. Navy de Estados Unidos de América para aplicaciones de gestión o administración. Con el paso de los años aparecieron nuevos lenguajes como **Pascal**, **Basic**, **C**, **C++**, **Ada**, **Java**, **C♯**, **HTML**, **XML**, entre otros.

Los lenguajes de programación de alto nivel se componen de un conjunto de instrucciones o primitivas más fáciles de escribir y recordar su función que los lenguajes máquina y ensamblador. Sin embargo, los programas escritos en un lenguaje de alto nivel, como C o Java necesitan ser traducidos a código máquina; para eso se requiere un programa denominado **traductor**. Estos programas de traducción se denominaron técnicamente, **compiladores**. De este modo, existen compiladores de C, Fortran, Pascal, Java, etcétera. También surgió una alternativa a los traductores compiladores como medio de implementación de lenguajes de tercera generación que se denominaron **intérpretes**. Estos programas eran similares a los traductores, excepto que ellos ejecutaban las instrucciones a medida que se traducían en lugar de guardar la versión completa traducida para su uso posterior. Es decir, en vez de producir una copia de un programa en lenguaje máquina que se ejecuta más tarde (éste es el caso de la mayoría de los lenguajes, C, C++, Pascal, Java), un intérprete ejecuta realmente un programa desde su formato de alto nivel, instrucción a instrucción. Cada tipo de traductor tiene sus ventajas e inconvenientes, aunque hoy día prácticamente los traductores utilizados son casi todos compiladores por su mayor eficiencia y rendimiento.

Sin embargo, en el aprendizaje de programación se suele comenzar también con el uso de los lenguajes algorítmicos, similares a los lenguajes naturales, mediante instrucciones escritas en pseudocódigo (o seudocódigo), que son palabras o abreviaturas de palabras escritas en inglés, español, portugués, etcétera. Posteriormente, se realiza la conversión al lenguaje de alto nivel que se vaya a utilizar realmente en la computadora, como C, C++ o Java. Esta técnica facilita la escritura de algoritmos como paso previo a la programación.

Cristian Pilapanta 1096211-2202090048

1.11. Paradigmas de programación

La evolución de los lenguajes de programación ha ido paralela a la idea de paradigma de programación: enfoques alternativos a los procesos de programación. En realidad, un paradigma de programación representa fundamentalmente enfoques diferentes para la construcción de soluciones a problemas y por consiguiente afectan al proceso completo de desarrollo de software. "Un paradigma de programación indica un método de realizar cómputos y la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa". Los paradigmas fundamentales están asociados a determinados modelos de cómputo. También se asocian a un determinado estilo de programación. Los lenguajes de programación suelen implementar, a menudo de forma parcial, varios paradigmas; un caso de éxito de lenguaje de programación multiparadigma es Python.

Los paradigmas de programación clásicos son: *procedimental* (*o imperativo*), *declarativo y orientado a objetos*, aunque existen otros paradigmas asociados a los sistemas concurrentes (programación concurrente) o tipados (*scripting*) como JavaScript.

Paradigma imperativo (procedimental)

El paradigma imperativo o procedimental representa el enfoque o método tradicional de programación. Un lenguaje imperativo es un conjunto de instrucciones que se ejecutan una por una, de principio a fin, de modo secuencial excepto cuando intervienen instrucciones de salto de secuencia o control. Este paradigma define el proceso de programación como el desarrollo de una secuencia de órdenes (comandos) que manipulan los datos para producir

los resultados deseados. Por consiguiente, el paradigma imperativo señala un enfoque del proceso de programación mediante la realización de un algoritmo que resuelve de modo manual el problema y a continuación expresa ese algoritmo como una secuencia de órdenes. En un lenguaje procedimental cada instrucción es una orden o secuencia de órdenes para que la computadora realice alguna tarea específica.

Los lenguajes imperativos tradicionales son C, Fortran, Basic, Pascal, Ada, aunque otros lenguajes actuales como C++, C♯, Java y Python incorporan propiedades de lenguajes imperativos.

Paradigma declarativo

En contraste con el paradigma imperativo, el paradigma declarativo solicita al programador que describa el problema en lugar de encontrar una solución algorítmica al problema; describe qué se debe calcular, sin explicitar el cómo. El control de flujo suele estar asociado a la composición funcional, la recursividad y/o técnicas de reescritura y unificación. El paradigma declarativo contiene, a su vez, otros paradigmas: paradigma lógico, funcional, reactivo y descriptivo.

El *paradigma lógico* utiliza el principio del razonamiento lógico para responder a las preguntas o cuestiones consultadas. Se basa en la lógica formal y en el cálculo de predicados de primer orden. El razonamiento lógico se basa en la deducción. El lenguaje de programación lógica por excelencia es Prolog.

El *paradigma funcional* se basa en los modelos de computo del cálculo Lambda. Modelos de lenguajes funcionales son: Lips, Haskell y Scheme.

El *paradigma reactivo* se basa en la teoría de grafos. Oz y Clojure son dos lenguajes de programación reactivos.

Cristian Pilapanta 1095211-2200050050

Paradigma orientado a objetos

El paradigma orientado a objetos se asocia con el proceso de programación llamado **programación orientada a objetos (POO)** que consiste en un enfoque totalmente distinto al proceso procedimental. El enfoque orientado a objetos guarda analogía con la vida real. El desarrollo de software **POO** se basa en el diseño y construcción de objetos que se componen a su vez de datos y operaciones que manipulan esos datos. El programador define en primer lugar los objetos del problema y a continuación los datos y operaciones que actuarán sobre esos datos. Las ventajas de la programación orientada a objetos se derivan esencialmente de la estructura modular existente en la vida real y el modo de respuesta de estos módulos u objetos a mensajes o eventos que se producen en cualquier instante.

Los orígenes de la POO se remontan a los **tipos abstractos de datos** como parte constitutiva de una estructura de datos.

C++, lenguaje orientado a objetos, por excelencia, es una extensión del lenguaje C y contiene las tres propiedades más importantes: *encapsulamiento, herencia y polimorfismo*. **Smalltalk** es otro lenguaje orientado a objetos muy potente y de gran impacto en el desarrollo del software orientado a objetos que se ha realizado en las últimas décadas.

Hoy día **Java** y **C♯** son herederos directos de C++ y C, y constituyen los lenguajes orientados a objetos más utilizados en la industria del software del siglo XXI. **Visual Basic** y **VB.Net** son otros lenguajes orientados a objetos, no tan potentes como los anteriores, pero extremadamente sencillos y fáciles de aprender. **Python** es un lenguaje de programación orientado a objetos, aunque uno de sus grandes éxitos es que es un lenguaje *multiparadigma*.

1.12. Internet y la Web

El origen de la actual red de Internet se remonta a la creación de la red ARPANET en noviembre de 1969 que conectaba a diferentes computadoras de la época, con objetivos estrictamente militares y con la finalidad de transmisión de datos entre computadoras conectadas. Victor Cerf y Bob Khan publicaron en 1974 el protocolo TCP/IP (protocolo de control de transmisión/protocolo de Internet) y fue el detonante para la expansión en la década de 1980 de la ya conocida como red Internet. Comienza a expandirse el correo electrónico, la mensajería instantánea, los sistemas de nombres de dominio (DNS). México fue el primer país hispano que tuvo conexión a Internet en 1989 y un año más tarde lo hicieron España, Argentina y Chile. En 1989, el investigador Tim Berners Lee del CERN suizo presentó un software basado en protocolos que permitían visualizar la información con el uso de hipertexto. Suele consi-

derarse a 1989 como el año de creación de la **Web** (World Wide Web) aunque fue unos años más tarde cuando se comenzó a expandir; asimismo, se considera a 1991 como el año del lanzamiento del lenguaje **HTML** (HyperText Markup Language), que se convertiría en los años posteriores en el estándar de diseño web. En 1994 se creó el **W3C** (World Wide Web Consortion), organismo mundial que gestiona la Web actual, aunque es la ISOC (Internet Society), la Sociedad Internet, creada en 1992, junto con InterNic y la ICANN (Internet Corporation for Assigned Names and Numbers) las instituciones que gestionan los nombres de dominio en la red.

La información en la Web se presenta en páginas que se entrelazan unas con otras en la telaraña universal que constituye la World Wide Web. Las páginas web residen en un sitio web que se identifica por su dirección, la URL (*Uniform Resource Locator*). La World Wide Web (*WWW*) o simplemente la Web fue creada en 1989 por Bernards Lee en el CERN (European Laboratory for Particles Physics), aunque su difusión masiva comenzó en 1993 como medio de comunicación universal. La Web es un sistema de estándares aceptados universalmente para almacenamiento, recuperación, formateado y visualización de información, utilizando una arquitectura cliente/servidor. Se puede utilizar la Web para enviar, visualizar, recuperar y buscar información o crear una página web. La Web combina texto, hipermedia, sonidos y gráficos, utilizando interfaces gráficas de usuario para una visualización fácil.

La navegación por Internet y a través de la Web se realiza mediante programas denominados navegadores web (*browsers*). Un navegador es una interfaz gráfica de usuario que permite "navegar" a través de la Web. Se utiliza el navegador para visualizar textos, gráficos y sonidos de un documento web y activar los enlaces (*links*) o conexiones a otros documentos. Cuando se hace clic (con el ratón o *mouse*) en un enlace a otro documento se produce la transferencia de ese documento situado en otra computadora a su propia computadora. Los navegadores más populares son: Explorer de Microsoft, Firefox de Mozilla, Chrome de Google, Safari de Apple y Opera.

La World Wide Web está constituida por millones de documentos enlazados entre sí, denominados páginas web. Una página web, normalmente, está construida por texto, imágenes, audio y video, al estilo de la página de un libro. Una colección de páginas relacionadas, almacenadas en la misma computadora, se denomina sitio web (*Website*). Un sitio web está organizado alrededor de una página inicial (*home page*) que sirve como página de entrada y punto de enlace a otras páginas del sitio. Cada página web tiene una dirección única, conocida como **URL**. Por ejemplo, la URL de la página inicial de este libro es: www.mhhe.com/latam/joyanes_fp5e

La Web se basa en un lenguaje estándar de hipertexto denominado HTML que da formatos a documentos e incorpora enlaces dinámicos a otros documentos almacenados en la misma computadora o en computadoras remotas. El navegador web está programado de acuerdo con el estándar citado. Los documentos HTML, cuando ya se han situado en Internet, se conocen como páginas web y el conjunto de páginas web pertenecientes a una misma entidad (empresa, departamento, usuario individual) se conoce como sitio web (*Website*). En los últimos años ha aparecido un nuevo lenguaje de marcación para formatos, heredero de HTML, y que se está convirtiendo en estándar universal: el lenguaje XML.

Otros servicios que proporciona la Web y ya muy populares para su uso en el mundo de la programación son: el correo electrónico y la mensajería instantánea. El correo electrónico (*e-mail*) utiliza protocolos específicos para el intercambio de mensajes: SMTP (*Simple Mail Transfer Protocol*), POP (*Post Office Protocol*) e IMAP (*Internet Message Action Protocol*). La mensajería instantánea o chat permite el diálogo en línea simultánea entre dos o más personas, y cuya organización y estructura han sido trasladadas a los teléfonos celulares, en los que también se puede realizar este tipo de comunicaciones con mensajes conocidos como "cortos" SMS (*short message*) o MMS (*multimedia message*).

1.12.1. Desarrollo de programas web

Una de las normas de desarrollo del software web es que sea sencillo y de tamaño reducido. La Web, como ya se ha comentado, se ha convertido en una plataforma de aplicaciones, desarrollo, entrega y ejecución. El escritorio web permite ejecutar aplicaciones en un navegador web, en un entorno similar al de un escritorio. El uso de la Web como plataforma forma parte de un movimiento dirigido a aplicaciones independientes del sistema operativo.

Por último, cabe citar otra característica notable que ha traído la Web 2.0: la beta perpetua y el desarrollo ágil. El ciclo de creación y publicación de versiones del software tradicional se está transformando. El desarrollo tradicional de software exigía pruebas exhaustivas y versiones beta para crear una versión definitiva. Hoy, la preocupación fundamental de la creación de software es el desarrollo de aplicaciones con versiones más frecuentes, es decir, el periodo de beta perpetua utilizando la Web como plataforma. Las actualizaciones se realizan en los servidores web en los que se almacena la aplicación y la distribución de software en CD se reduce a la mínima expresión.

Cristian Pilapanta 1095211-2202090050

1.13. *Cloud computing* (computación en la nube como servicio)

Desde mediados de 2008 una nueva arquitectura o paradigma de computación ha nacido y se está implantando de modo acelerado: *cloud computing* (computación en la nube). ¿Qué es la computación en la nube? Este concepto, que en realidad no es nuevo, es una filosofía de trabajo en la computación y en la informática que proviene de ofrecer el hardware, el software, los programas como servicios, al igual que sucede con cualquier otro servicio común, como la electricidad, el gas, el teléfono, etc. La computación en la nube es un nuevo modelo que ofrece a usuarios, programadores y desarrolladores, la posibilidad de ejecutar los programas directamente, sin necesidad de instalación, ni mantenimiento. Éste es el caso de las aplicaciones actuales de la Web en la que el usuario descarga el programa y a continuación lo ejecuta. Este es el caso de Google Maps, Google Street View, las redes sociales, los blogs, etc., o los casos comunes del correo electrónico web como Gmail, Yahoo!, etc., en el que el usuario sólo debe ejecutar los programas que se encuentran en la nube (*cloud*), los infinitos sistemas de computadoras y centros de datos existentes en el mundo, en los que hay numerosos servicios (programas, sistemas operativos, redes de comunicaciones, todos ellos "virtuales") procedentes de numerosos proveedores que ofrecen estos servicios ya sea gratuitos o bien con pago de una tasa diaria, semanal, mensual, etc. Cada día hay más proveedores de servicios en la nube, aunque los más conocidos son: Google, Amazon, Microsoft, IBM, Salesforce, etcétera.

Esta nueva arquitectura y filosofía de computación requiere nuevos conceptos para el desarrollo de programas, centrados fundamentalmente en servicios web, aplicaciones web que residirán en la nube de computadoras, y que los usuarios descargarán y ejecutarán de manera inmediata. De una manera simple, la computación en la nube *(cloud computing)* es una tecnología que permite acceso remoto a software, almacenamiento de archivos y procesamiento de datos por medio de Internet, siendo así, una alternativa a la ejecución en una computadora personal o servidor local. En el modelo de nube, no hay necesidad de instalar aplicaciones localmente en computadoras. La computación en la nube ofrece a los individuos y a las empresas la capacidad de un *pool* de recursos de computación con buen mantenimiento, seguro, de fácil acceso y bajo demanda.

Cloud computing es una colección grande de servidores distribuidos en centros de datos alojados en diferentes lugares del mundo y que proporcionan servicios de Internet. Los distribuidores de la nube son los que ofrecen los diferentes servicios que contratan a los clientes y que éstos alquilan como servicios.

Los sistemas de computadoras han de entregar funcionalidad a sus usuarios. Los usuarios de computadoras y las empresas cliente no se preocupan en principio de cómo organizar el almacenamiento en terabytes y la velocidad del procesador en GHz, sino que contratan los servicios y en las cantidades que requieran y consideran de interés a nivel personal, profesional o empresarial.

Muchas empresas continúan sin migrar, todavía a la nube, y disponen de sus propios centros de datos, aunque cada vez con mayor frecuencia están migrando a la nube y externalizando sus servicios en empresas de distribución de la nube especializadas. La contratación de servicios de la nube requiere la contratación legal oportuna y el cumplimiento de los denominados acuerdos de nivel de servicios (SLA).

La computación en la nube es una tendencia que lleva bastantes años y que, en esencia, es una plataforma de computación virtual que está en Internet. Una plataforma de la nube se define en términos de servicios que proporciona en lugar de aportar configuración física organizacional. Las computadoras y el almacenamiento aparecen al usuario en una entidad que se llama la nube, dotada de grandes servidores físicos instalados en centros de datos que disponen a lo largo del mundo los distribuidores de la nube, como son Amazon, Google, IBM, HP, Oracle, o Rackspace. El almacenamiento de datos se localiza en los centros de datos distribuidos a lo largo del mundo y, normalmente, no están conectados a un servidor físico que se pueda manejar por el cliente, sino que están situados en un centro de datos (normalmente, se desconocerá inclusive su ubicación física) y el software del sistema está diseñado para la configuración seleccionada, pero se presenta como una máquina virtual al usuario.

Los servicios ofrecidos por la nube (cloud computing) son: **Software como Servicio** (**SaaS**, Software as a Service), **Plataformas como servicio** (**PaaS**, Platform as a Service) e **Infraestructuras como servicios** (**IaaS**, Infrastructure as a Service).

1.13.1. Software como servicio (SaaS)

El software como servicio (Saas, *Software as a Service*) es un tipo de software que se descarga de un servidor web y no necesita instalación en el equipo del cliente, sólo su ejecución. Este tipo de modelo de software se apoya en la

Cristian Pilapanta 1095211-2202090051

nube. En los servidores del distribuidor de software correspondiente se almacenan los diferentes programas y aplicaciones; el usuario selecciona aquel que desea utilizar y sólo necesita ejecutarlo. El usuario paga una cuota mensual, anual o por cualquier otro periodo, o bien la descarga y ejecución del usuario es gratis. Existen numerosas aplicaciones de software como servicio, entre ellos destacan *Windows Office 365*, el programa clásico de ofimática de Windows, pero que se puede ejecutar desde la nube y no requiere instalación, y *Saleforce.com*, un programa de gestión y administración con los clientes que está configurando un nuevo tipo de desarrollo de aplicaciones de software y unos nuevos modelos de negocios y de aplicaciones comerciales.

Cada día es más frecuente el uso de software como servicio, sobre todo en pequeñas y medianas empresas, y en usuarios finales. En la computación o informática móvil, este tipo de software es todavía más utilizado, ya que existen miles de aplicaciones de software que no requieren más que la descarga y ejecución por parte del usuario; éste es el caso de numerosos videojuegos, programas de gestión, programas didácticos, etcétera.

El proveedor de la nube proporciona aplicaciones completas en Internet sin necesidad de instalar ningún componente en sus computadoras. El consumidor del servicio no mantiene la aplicación y sólo la utiliza y no necesita preocuparse de su infraestructura. Ejemplos muy conocidos son: Gmail, Dropbox, GotoMeeting, Netflix, Spotify etc. O software especializado en productos de administración de sistemas de información en empresas, como CRM o ERP del distribuidor acreditado Salesforce.

Una desventaja del SaaS es que el consumidor tiene poco control sobre el comportamiento del producto y puede ser problemático si la compañía proveedora del servicio cambia sus procesos, políticas de distribución o tiene fallos en sus propias instalaciones. El usuario utiliza programas ejecutables.

Cristian Pilapanta 1095211-2202090053

1.13.2. Plataforma como servicio (PaaS)

PaaS proporciona hardware de servidores, sistemas operativos, servicios de bases de datos, componentes de seguridad, servicios de recuperación y backup, o lenguajes de programas y entornos integrados de desarrollo (EID). El proveedor gestiona el desempeño y la disponibilidad del entorno, mientras que el cliente gestiona las aplicaciones hospedadas en la nube. El cliente normalmente contrata almacenamiento, gigabytes o terabytes, procesadores a utilizar o velocidad de transmisión de datos. Proveedores conocidos: Google App Engine, AWS de Amazon, Microsoft Azure Cloud Services, etcétera.

1.13.3. Infraestructura como servicio (IaaS)

Cuando se requieren cambios rápidos en las configuraciones se han de gestionar sistemas operativos, software de gestión de bases de datos, etc. Se necesitan máquinas virtuales, con megabytes de almacenamiento y de transferencia de datos: Amazon EC2, Google Compute Engine, Microsoft Azure Services Platform, Rackspace, HP Cloud.

Los proveedores proporcionan elasticidad: capacidad para añadir y eliminar recursos bajo demanda (un cliente sólo paga tanta infraestructura como necesite, de modo que se puede contratar capacidad extra durante periodos pico o temporalmente). Esta flexibilidad puede ahorrar a una compañía grandes costes cuando tiene grandes variaciones de demandas de computación.

El almacenamiento en la nube es una extensión de IaaS para obtener pequeñas o grandes cantidades de almacenamiento a través de servicios (en este tipo de servicio, en numerosas ocasiones, gratuitos para condiciones determinadas) como Dropbox, Google Drive, Apple Store, Amazon Cloud Drive, etcétera.

En el lado del consumidor los arquitectos de software y los programadores deben ser conscientes de los modelos de cloud a contratar en función de los recursos necesarios

Infraestructura como servicio (IaaS)
Un proveedor proporciona a los clientes acceso de pago por uso al almacenamiento, las redes, los servidores y otros recursos informáticos en el *cloud*.
Plataforma como servicio (PaaS)
Un proveedor de servicios ofrece acceso a un entorno basado en *cloud* en el cual los usuarios pueden crear y distribuir aplicaciones. El proveedor proporciona la infraestructura subyacente.

Software como servicio (SaaS)
Un proveedor de servicios proporciona el software y las aplicaciones a través de Internet. Los usuarios se suscriben al software y acceden a él a través de la Web o las APIs (interfaz de programación de aplicaciones) del proveedor.
https://www.ibm.com/es-es/cloud/learn/iaas-paas-saas

1.14. Internet de las cosas

La tecnología **Internet de las cosas** (*Internet of Things*, **IoT**)) se ha ido asentando y popularizando en los últimos años gracias a la implantación de tecnologías como RFID (identificación por radiofrecuencia), Bluetooth, NFC (proximidad), sensores y tecnologías móviles o inalámbricas como 5G, WiFi, NB-IoT, LPWAN. Todas estas tecnologías han hecho que cada día más existan millones de cosas que transmiten información en tiempo real y bajo demanda, entre sí y a otras cosas.

Internet de las cosas o Internet de los objetos, como también comienza a conocerse, es una tendencia emergente e imparable y que se refiere al hecho de que miles de cosas u objetos de todo tipo (libros, zapatos, electrodomésticos, vehículos, trenes, sensores) se pueden conectar entre sí y, a su vez, conectarse a Internet para intercambiar información de todo tipo entre ellos mismos, es decir, los objetos se interconectan entre sí. Como a los objetos se les puede asociar una dirección de Internet (necesaria para su conexión a la red) IP, se podrán formar grandes redes de objetos. Como el nuevo protocolo IPv6 va a permitir miles de millones de direcciones IP, eso significa que miles de millones de objetos podrán tener asociada una dirección IP y por consiguiente, conectarse con Internet y con otros objetos en tiempo real.

En esencia, el Internet de las cosas se está refiriendo a la integración de sensores, chips RFID, chips NFC, en dispositivos y objetos cotidianos que quedarán conectados a Internet a través de las redes fijas o inalámbricas. Serán fácilmente integrables en hogares, entornos de trabajo y lugares públicos. Cualquier objeto podrá conectarse con otro objeto y a su vez ambos a Internet.

Esta nueva tendencia tecnológica exigirá la programación de aplicaciones de todo tipo que deberán escribir programadores para atender a esta nueva demanda social y tecnológica. La programación del futuro, ya casi presente, se verá muy afectada por el Internet de las cosas, debido a que todas tendrán conexión a Internet en cualquier momento y lugar. El mundo conectado en que vivimos no sólo conectará a personas, sino que conectará a esas personas con miles de millones de objetos a través de Internet.

1.15. Big Data. Los grandes volúmenes de datos

Big Data (grandes datos, grandes volúmenes de datos o *macro datos* como recomienda utilizar la Fundación Fundéu BBVA, Fundación del español urgente) supone la confluencia de una multitud de tendencias tecnológicas que venían madurando desde la primera década del siglo XXI, y que se consolidaron durante los años 2011 a 2013, cuando detonaron e interrumpieron con gran fuerza en organizaciones y empresas, en particular, y en la sociedad, en general: movilidad, redes sociales, aumento de la banda ancha y reducción de su costo de conexión a Internet, medios sociales (en particular las redes sociales), Internet de las cosas, geolocalización, y de modo muy significativo la computación en la nube (*cloud computing*).

Una definición muy significativa es del McKinsey Global Institute, que en un informe muy reconocido y referenciado, de mayo de 2011, define el término del siguiente modo: "Big Data se refiere a los conjuntos de datos cuyo tamaño está más allá de las capacidades de las herramientas típicas de software de bases de datos para capturar, almacenar, gestionar y analizar". El término Big Data en realidad ha surgido debido esencialmente al crecimiento exponencial de los datos, especialmente de datos no estructurados (los datos no guardados en las bases de datos tradicionales) como datos procedentes de redes sociales, de sensores, de la biometría, audio, video, fotografía, etcétera.

Big Data se caracteriza por sus tres grandes dimensiones conocidas como modelo de las tres V (3 V o V3): volumen, velocidad y variedad (*variety*). El volumen se refleja hoy día en petabytes y exabytes. En 2000, se almacenaron en el mundo 800 000 petabytes y se espera que en 2020 se alcancen los 40 zettabytes (ZB). Sólo Twitter genera más de 12 terabytes (TB) de datos cada día, Facebook, 15 TB; los servidores de Google 1 petabyte cada hora, y al-

Cristian Pilapanta 1096211-2202000264

gunas empresas generan terabytes de datos cada hora cada día del año. La importancia de la velocidad de los datos o el aumento creciente de los flujos de datos en las organizaciones junto con la frecuencia de las actualizaciones de las grandes bases de datos son características importantes a tener en cuenta. Esto requiere que su procesamiento y posterior análisis, normalmente, ha de hacerse en tiempo real para mejorar la toma de decisiones sobre la base de la información generada. Las fuentes de datos son de cualquier tipo. La variedad se manifiesta en que los datos pueden ser estructurados y no estructurados (texto, datos de sensores, audio, video, flujos de clics, archivos log), y cuando se analizan juntos se requieren nuevas técnicas. Imaginemos el registro en vivo de imágenes de las cámaras de video de un estadio de fútbol o de vigilancia de calles y edificios. En los sistemas de Big Data las fuentes de datos son diversas y no suelen ser estructuras relacionales típicas. Los datos de redes sociales, de imágenes pueden venir de una fuente de sensores y no suelen estar preparados para su integración en una aplicación.

1.16. Los lenguajes de programación más populares: índice TIOBE

El índice TIOBE (`http://pypl.github.io/PYPL.html`) es uno de los *rankings* más acreditados sobre lenguajes de programación. El índice de la comunidad de programación TIOBE es un indicador de la popularidad de lenguajes de programación. Se publica cada mes y su lema central es: "El índice se puede utilizar para verificar si sus destrezas de programación están todavía actualizadas hasta la fecha o tomar decisiones estratégicas acerca del lenguaje de programación que debe adoptarse cuando comience a construir un nuevo sistema de software". El índice se obtiene del estudio y análisis de datos seleccionados de 25 motores de búsqueda y sitios web especializados en programación de computadoras, entre ellos el buscador Alexa.

En enero de 2020, el *ranking* de los 10 lenguajes de programación más utilizados por las empresas reconoce el crecimiento en popularidad del lenguaje Python y la confirmación de que los lenguajes de programación C, C++, Java y C# siguen siendo los más reconocidos (figura 1.13).

Julio 2019	Julio 2018	Change	Proramming Lenguage	Rating
1	1		Java	15.058%
2	2		C	14.21%
3	4	⋀	Python	9.260%
4	3	⋁	C++	6.705%
5	6	⋀	C#	4.365%
6	5	⋁	Visual Basic.NET	4.208%
7	8	⋀	Java Scipt	2.304%
8	7	⋁	PHP	2.167%
9	9		SQL	1.977%
10	10		Objective-C	1.686%

Figura 1.13. *Ranking* de los 10 lenguajes de programación más populares (Índice TIOBE, enero 2020).

En los puestos 11 al 20 aparecen nuevos lenguajes de programación como Ruby, Perl, Go, R y también lenguajes de programación tradicionales como Visual Basic, Objective-C o Delphi/Object Pascal y el lenguaje Ensamblador o el lenguaje para programación de matemáticas MATLAB.

Otros índices acreditados como PYPL (*Popularity of Programming Languages Index* PYPl.github.io/PYPL.html) y recogidos en el prestigioso sitio de estadísticas Statista, en enero de 2020 publicaba que el lenguaje de programación más utilizado en esa fecha era Python, con un porcentaje de 29.72%, seguido de Java con 19.03% y, ya con porcentajes inferiores a 8%, JavaScript, C#, PHP, C/C++, R, Objective-C, Swift, Matlab, TypeScript, Kotlin, Ruby, Go, Scala y Visual Basic.

Cristian Pilapanta 1095211-2202090055

1.17. Nacimiento de la programación moderna: lenguajes de programación de referencia (C, C++, Java, Python y C#)

El índice TIOBE relativo al ranking de programas más populares y utilizados por programadores, desarrolladores y distribuidores de software recoge como los cinco primeros lenguajes de programación a Java, C, Python, C++ y C#. En este apartado y con el objeto de que el lector las conozca, se describen las características más notables de cada lenguaje y sus aplicaciones y sectores de mayor impacto.

C

La creación de C supuso una revolución en el mundo de la programación, como posteriormente supuso Java. Fue un resultado directo de la necesidad de disponer de un lenguaje de alto nivel, estructurado y eficiente que pudiera reemplazar al código ensamblador cuando se necesitaban crear programas del sistema.

Antes de C, los programadores tenían que elegir entre lenguajes clásicos como Cobol, Fortran y Basic posteriormente aparecieron Pascal y Object Pascal. Fortran se utilizaba para escribir programas eficientes para aplicaciones científicas, aunque era complejo de codificar. Cobol era el lenguaje por excelencia para aplicaciones de negocios y empresariales. Basic era un lenguaje fácil de aprender, pero no muy potente, ya que carecía de estructura y no era muy adecuado para realizar programas grandes. Los tres lenguajes no estaban diseñados para programación estructurada. Posteriormente, como hemos comentado, nació Pascal, ya estructurado, fácil de aprender, pero con fallos en un amplio rango de programas.

El lenguaje ensamblador, alternativa a todos ellos, permite la realización de programas más eficientes, pero no es fácil de aprender ni de utilizar con eficiencia; además, la depuración del código ensamblador suele ser bastante difícil.

C fue la evolución lógica para desarrollar un lenguaje de programación estructurado, completo y eficiente. Fue creado por Dennis Ritchie y su primera implementación se ejecutó sobre una computadora DEC PDP-11 con un sistema operativo UNIX. Se desarrolló en la década de 1970 y fue de facto el estándar de programación; su lanzamiento fue la publicación del libro *The C Programming Language* (Prentice-Hall, 1978) por Brian Kernighan y Dennis Ritchie. C fue formalmente estandarizado en diciembre de 1989 por el American National Standards Institute (ANSI) y adoptado universalmente.

La creación de C está considerada como el principio de la era moderna de los lenguajes de programación de computadoras. El resultado fue un lenguaje potente, eficiente y estructurado que es relativamente fácil de aprender y que hoy, al comienzo de la tercera década del siglo XXI sigue siendo uno de los más utilizados, ya que fue diseñado pensando en la eficiencia de los programadores. Java, del que posteriormente hablaremos, heredó muchas de esas propiedades.

C++

C era el lenguaje por excelencia en la década de 1980, pero era difícil de gestionar cuando se necesitaban programas complejos. C++ fue la solución apoyándose en técnicas orientadas a objetos que ayudaban a organizar problemas complejos a través del uso de características como encapsulamiento, herencia y polimorfismo ya comentadas anteriormente, lo que facilitó la resolución de problemas complejos mediante la descomposición en módulos y el concepto de clases.

C++ fue creado por Bjarne Stroustrup que le llamó "lenguaje C con clases". En 1983 cambió el nombre a C++ y le añadió características orientadas a objetos. En 1985, Stroustrup publicó su libro *The C++ Programming Language*[6] que describía el lenguaje C++. En 1998 fue estandarizado por ANSI, como ANSI C++. En 2012 se publicó el estándar C++11. En 2014 se lanzó la última versión estándar, la nueva *C++14 standard*. Ambas versiones fueron estandarizas también por la International Organization for Standarization (ISO) y la International Electrotechnical Comission (IEC). C++ es un lenguaje híbrido cuya última versión estables es C++17. *1 de diciembre de 2017* (*ISO/IEC 14882:2017*). Última versión en pruebas C++20 (2020).

[6] El libro fue traducido por el autor de este libro, Luis Joyanes, con el título *El lenguaje de programación* C++ y publicado en la editorial Addison-Wesley (2002).

Cristian Pilapanta 1095211-2202090058

Java

Las dos características más sobresalientes de Java son: seguridad y portabilidad. La clave para direccionar ambas características es que la salida de un compilador de Java no es código ejecutable, sino bytecode. Bytecode es un conjunto de instrucciones altamente optimizadas diseñadas para ser ejecutadas en la denominada máquina virtual de Java (JVM, Java Virtual Machine). En esencia, el JVM fue creado como "un intérprete" de bytecode. Antes de que se pueda compilar y ejecutar un programa Java se debe tener instalado un entorno de programación JDK.

Los orígenes de Java

Java fue creado por James Gosling, Patrick Naughton, Crist Warth, Ed Frank y Mike Sheridan en Sun Microsystems en 1991. El lenguaje se llamó inicialmente "Oak" y se renombró a Java en 1995. Curiosamente como señalan sus autores su principal motivación no fue Internet, por aquel entonces con la W3 emergente, sino que fue la necesidad de crear un lenguaje independiente de la plataforma que se pudiera utilizar para crear software que se pudiera embeber en diferentes dispositivos electrónicos de consumo como hornos de microondas, tostadoras o dispositivos con mando a distancia (como los de control de televisión). En aquella época muchos tipos diferentes de CPUs se utilizaban como controladores. El problema era que, en ese momento, la mayoría de los lenguajes de computadoras estaban diseñados para compilarse en un código máquina destinado a un tipo específico de CPU, por ejemplo, C y C++. Sin embargo, a medida que avanzaba la evolución de Java, Internet y la Web jugaron un rol crucial en Java.

Java estuvo influenciado por C y C++. Java heredó su sintaxis principalmente de C, y de C++ su modelo de objetos. La relación de Java con C y C++, como señala Schildt (2019, p. 15) es importante por numerosas razones:

1. La creación de Java: muchos programadores estaban familiarizados con la sintaxis de C/C++.
2. Los diseñadores de Java no querían "reinventar la rueda", ya que la evolución de la programación moderna comenzó con C, se movió a C++ y luego a Java.
3. Java no es una versión mejorada de C++ ni fue creado para reemplazarlo. Java fue diseñado para resolver un determinado número de problemas y no pensando en Internet ni en C++.

Internet ayudó a catapultar Java en programación y, a su vez, Java tuvo un profundo efecto en Internet, ya que dirigió a muchos programadores a la Web. Asimismo, Java innovó un nuevo tipo de programa en red llamado *applet* que cambió el mundo online y sus contenidos. Por último, Java potenció temas asociados a Internet, especialmente portabilidad y seguridad.

Un applet de Java es un tipo especial de programa (Schildt, p. 16) Java que se diseña para ser transmitido por Internet y ejecutado automáticamente en el interior de un navegador web compatible con Java. Si el usuario hace clic en un enlace que contiene un applet, éste se descargará y ejecutará en el navegador automáticamente. Los applets fueron concebidos como pequeños programas, normalmente utilizados para visualizar datos proporcionados por el servidor, manejar la entrada del usuario o proporcionar funciones sencillas tales como un calculador de préstamos (*loan calculator*). La característica clave de los applets es que se ejecutan localmente en lugar de en el servidor. En esencia, los applets permitían que alguna funcionalidad se moviera del servidor al lado del cliente.

Java 8 apareció en 2014, Oracle lanza la nueva versión JDK 9. **Java 11** inicia una nueva era en las licencias de uso. Hasta ahora se podía descargar y programar con el kit de Desarrollo de Java oficial de Oracle y luego poner la aplicación en producción y distribuirla sin tener que pagar por ello. Java 11 se puede seguir desarrollando pero habrá que pagar una licencia a Oracle si se desea utilizar para poner aplicaciones en producción: 2.5 dólares al mes por cada usuario de escritorio y 2.5 dólares al mes por procesador en caso de aplicación de servidor. JDK con Java ya no es gratuita (OpenJDK). Se debe pagar una licencia mensual por usuario o por procesador si se quiere usar en producción. En marzo de 2019 se presentó la versión Java 12 y en septiembre de 2019, la versión Java 13.

Python

Surgió a principio de los 90, desarrollado inicialmente por Guido van Rossum, ingeniero holandés del CWI de Amsterdam (Centro de Investigación en Ciencias de la Computación de Holanda), y presentado en diciembre de 1989. Su nombre Python, cuenta su historia, proviene del famoso grupo cómico británico Monty Python del que Guido era un gran fan. Python nació como un proyecto de software libre y hoy está gestionado por la Python Software Foundation. Está licenciado bajo PSFL, derivada de BSD, y compatible con GPL.

Cristian Pilapanta 1095211-2202090058

Es un lenguaje más fácil de aprender que los lenguajes de programación populares como C, C++, Java y C#. Es más fácil de leer que otros lenguajes de programación y muy utilizado en el sector de educación.

Python es un lenguaje de programación interpretado cuya filosofía es hacer resaltar una situación que favorezca la lectura de dicho código. Un lenguaje interpretado necesita de un programa intérprete que realice la traducción a medida que sea necesario, normalmente, instrucción a instrucción y generalmente no guardan el resultado de dicha traducción. Python es un lenguaje multiparadigma, es decir, soporta diferentes tipos de estructuración: programación orientada a objetos, imperativa-procedimental y en menor medida funcional. Además, es multiplataforma y usa un tipado dinámico, es decir, una variable puede cambiar su tipo de valor a lo largo de un programa.

Es un lenguaje muy demandado y la demanda de empleo de profesionales expertos en programación de Python es muy grande. Python es un lenguaje de programación muy utilizado para desarrollo de aplicaciones en Big Data, en ciencia de datos (*data science*), inteligencia artificial (desarrollo de algoritmos en aprendizaje automático *machine learning* y aprendizaje profundo *deep learning*) y en Internet de las Cosas.

Python 1.0 se lanzó en 1994, Python 2.0 en 2000 y Python 3.0 en 2008. La versión de Python 3.7.2 se lanzó el 24 de diciembre de 2018. El 18 de diciembre de 2019 se lanzó la última versión estable, Phyton 3.8.1.

Cristian Pilapanta 1095211-2020000259

Características destacadas

- *De código abierto (open source),* libre y ampliamente disponible con una comunidad masiva de programadores y desarrolladores. Existen numerosas aplicaciones de Python gratuitas y de código abierto.
- *Multiparadigma* (orientado a objetos, estructurado y funcional).
- *Multiplataforma* (originalmente se desarrolló en Unix); es compatible con cualquier sistema operativo que integre un intérprete programado.
- *Interpretado* (o de *script*) utiliza un intérprete. El código fuente se traduce a un pseudocódigo máquina intermedio llamado *bytecode*, la primera vez que se ejecuta, genera archivos `.pyc` o `.pyv` que son ejecutables. Los lenguajes interpretados son más flexibles y portables frente a los compilados que, por el contrario, tienen una ejecución más rápida.
- *Interactivo.* Dispone de un intérprete por la línea de comandos en el que se pueden introducir sentencias. Cada sentencia se ejecuta y produce un resultado visible que facilita la interpretación y probar los resultados de la ejecución rápidamente.
- *Funciones y librería.* Dispone de muchas funciones incorporadas en el propio lenguaje, para el tratamiento de cadenas de texto, números, archivos, etcétera. Además, existen muchas librerías de las que se pueden importar a un programa, incluso crear archivos comprimidos `.zip`.
- *Sintaxis clara y legible.* Es una sintaxis visual gracias a una notación indentada (con sangrado de texto mediante tabulación o espacios) de obligado cumplimiento. En lugar de palabras reservadas como *begin-end*, llaves se utilizan los espacios en Python para separar las porciones de código. Se debe tabular hacia dentro colocando un margen al código que iría dentro de una función o bucle. Esta característica facilita que todos los programadores adopten unas mismas notaciones y que los programas tengan aspectos similares.
- *Desarrollo web.* Muy popular en el desarrollo web (aplicaciones como Django, Dropbox, YouTube, Reddit, Instagram, Quora, Netflix utilizan Python).

C#

Unos años después de la creación de Java, Microsoft desarrolló el lenguaje C#, combinando propiedades de C++ y de Java, especialmente. De hecho, muchas características de C# descienden directamente de Java y ambos soportan propiedades de C++, en particular las propiedades de objetos y distribuidos. El aprendizaje de C# es relativamente fácil caso de conocer Java.

Resumen

Una computadora es una máquina para procesar información y obtener resultados en función de unos datos de entrada.

Hardware: parte física de una computadora (dispositivos electrónicos).

Software: parte lógica de una computadora (programas).

Existen cuatro tipos fundamentales de software: de sistemas, de aplicaciones, software propietario y software de código abierto *(open source)*.

Las computadoras se componen de:

- Dispositivos de entrada/salida (E/S).
- Unidad central de proceso (unidad de control y unidad lógica y aritmética) o **procesador**.
- Memoria principal, memorias caché y secundaria.
- Sistemas o canales de comunicación

Dispositivos de almacenamiento masivo de información (memoria auxiliar o externa) cada uno de los cuales tiene sus propias características y funcionalidades, y normalmente son más fáciles de transportar a máquinas diferentes que los escritos en lenguajes de bajo nivel.

Los lenguajes de programación en las computadoras se clasifican en tres grandes grupos: máquina, ensamblador y alto nivel. Desde el punto de vista de traducción los lenguajes de programación pueden ser compilados, interpretados o con características híbridas, y los programas traductores, compiladores e intérpretes.

Los programas escritos en lenguaje de alto nivel deben ser traducidos por un compilador antes de que se puedan ejecutar en una máquina específica. En la mayoría de los lenguajes de programación se requiere un compilador para cada máquina en la que se desea ejecutar programas escritos en un lenguaje específico.

Los lenguajes de programación se clasifican en:

- *Alto nivel*: Python, R, Visual Basic, C, C++, C♯, etc.
- *Bajo nivel*: Ensamblador.
- *Máquina*: Código máquina.

Los programas traductores de lenguajes son:

- *Compiladores*.
- *Intérpretes*.

Los cinco lenguajes de programación más utilizados según el *ranking* del índice TIOBE, edición de enero 2020, son: Java, C, Python, C++ y C♯.

Cristian Palapanta 1095211-2202060101

CAPÍTULO 2

Metodología de la programación y desarrollo de software

2.1. Fases en la resolución de problemas
2.2. Metodología de la programación
2.3. Programación estructurada
2.4. Programación orientada a objetos
2.5. Concepto y características de algoritmos
2.6. Escritura de algoritmos
2.7. Representación gráfica de los algoritmos
2.8. Herramientas y entornos de desarrollo de programación
ACTIVIDADES DE APRENDIZAJE
ACTIVIDADES COMPLEMENTARIAS
RESUMEN
EJERCICIOS

Cristian Pliqueria 1095211-2202090103

1095211220209010

INTRODUCCIÓN

Este capítulo le introduce a la metodología necesaria para la resolución de problemas con computadoras.

La resolución de un problema con una computadora se hace escribiendo un programa que exige al menos los siguientes pasos:

1. Definición o análisis del problema.
2. Diseño del algoritmo.
3. Transformación del algoritmo en un programa.
4. Ejecución y validación del programa.

Uno de los objetivos fundamentales de este libro es el *aprendizaje y diseño de los algoritmos*. Este capítulo introduce al lector en el concepto de algoritmo y de programa; además, le permite conocer las herramientas con las que el usuario puede "dialogar" con la máquina: *los lenguajes de programación*.

2.1. Fases en la resolución de problemas

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución. Aunque el proceso de diseñar programas es, esencialmente, un proceso creativo, se puede considerar una serie de fases o pasos comunes que generalmente deben seguir todos los programadores.

Las fases de resolución de un problema con computadora son:

- *Análisis del problema.*
- *Diseño del algoritmo.*
- *Codificación.*
- *Compilación y ejecución.*
- *Verificación.*
- *Depuración.*
- *Mantenimiento.*
- *Documentación.*

Las características más sobresalientes de la resolución de problemas son:

- ***Análisis.*** El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por la persona que encarga el programa.
- ***Diseño.*** Una vez analizado el problema, se diseña una solución que conducirá a un *algoritmo* que resuelva el problema.
- ***Codificación (implementación).*** La solución se escribe en la sintaxis del lenguaje de alto nivel (por ejemplo, C/C++) y se obtiene un programa fuente que se compila a continuación.
- ***Ejecución, verificación y depuración.*** El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores (denominados *bugs*, en inglés) que puedan aparecer.
- ***Mantenimiento.*** El programa se actualiza y modifica cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.
- ***Documentación.*** Escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera fase (*codificación*) se *implementa*[1] el algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas desarrolladas en las fases de análisis y diseño.

Las fases de *compilación* y *ejecución* traducen y ejecutan el programa. En las fases de *verificación* y *depuración*, el programador busca errores de las etapas anteriores y los elimina. Comprobará que mientras más tiempo se gaste en la fase de análisis y diseño, menos se gastará en la depuración del programa. Por último, se debe realizar la *documentación del programa*.

Antes de conocer las tareas a realizar en cada fase, se considera el concepto y significado de la palabra **algoritmo**. La palabra *algoritmo* se deriva de la traducción al latín de la palabra *Alkhô-warîzmi*,[2] nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX. Un **algoritmo** es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

Características de un algoritmo

- *Preciso* (indica el orden de realización en cada paso).
- *Definido* (si se sigue dos veces, obtiene el mismo resultado cada vez).
- *Finito* (tiene fin, un número determinado de pasos).

[1] En la 21a. edición del DRAE (Diccionario de la Real Academia Española) se ha aceptado el término *implementar*: (Informática) "Poner en funcionamiento, aplicar métodos, medidas, etc., para llevar algo a cabo".

[2] Escribió un tratado matemático famoso sobre manipulación de números y ecuaciones titulado *Kitab al jabr w'almuqabala*. La palabra álgebra se derivó, por su semejanza sonora, de *al-jabr*.

Cristian Pilapanta 1005211-2202000104

Un algoritmo debe producir un resultado en un tiempo finito. Los métodos que utilizan algoritmos se denominan *métodos algorítmicos*, en oposición a los métodos que implican algún juicio o interpretación, que se denominan *métodos heurísticos*. Los métodos algorítmicos se pueden *implementar* en computadoras; sin embargo, los procesos heurísticos no han sido convertidos fácilmente en las computadoras. En los últimos años las técnicas de inteligencia artificial han hecho posible la *implementación* del proceso heurístico en computadoras.

Ejemplos de algoritmos son: instrucciones para montar en una bicicleta, hacer una receta de cocina, obtener el máximo común divisor de dos números, etcétera. Los algoritmos se pueden expresar por *fórmulas*, *diagramas de flujo* o **N-S** y *pseudocódigos*. Esta última representación es la más común para su uso con lenguajes estructurados como C.

2.1.1. Análisis del problema

La primera fase de la resolución de un problema con computadora es el *análisis del problema*. Esta fase requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución deseada. Dado que se busca una solución por computadora, se precisan especificaciones detalladas de entrada y salida. La figura 2.1 muestra los requisitos que se deben definir en el análisis.

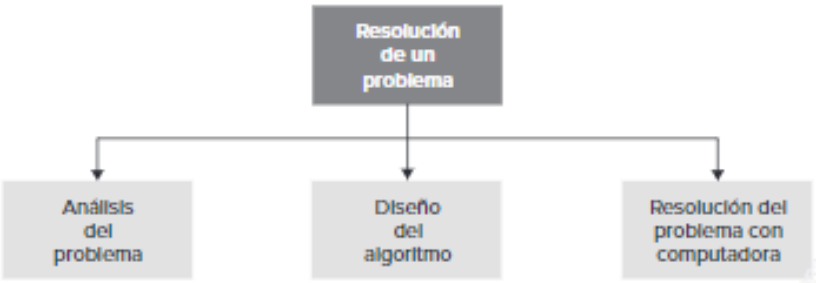


Figura 2.1. Análisis del problema.

Para poder identificar y definir bien un problema es conveniente responder a las siguientes preguntas:

- ¿Qué entradas se requieren? (tipo de datos con los cuales se trabaja y cantidad).
- ¿Cuál es la salida deseada? (tipo de datos de los resultados y cantidad).
- ¿Qué método produce la salida deseada?
- Requisitos o requerimientos adicionales y restricciones a la solución.

PROBLEMA 2.1

Se desea obtener una tabla con las depreciaciones acumuladas y los valores reales de cada año de un automóvil comprado por 20000 euros en el año 2005, durante los seis años siguientes suponiendo un valor de recuperación o rescate de 2000. Realizar el análisis del problema, conociendo la fórmula de la depreciación anual constante D para cada año de vida útil.

$$D = \frac{\text{costo} - \text{valor de recuperación}}{\text{vida útil}}$$

$$D = \frac{20000 - 2000}{6} = \frac{18000}{6} = 3000$$

$$\text{Entrada} \begin{cases} \text{costo original} \\ \text{vida útil} \\ \text{valor de recuperación} \end{cases}$$

Cristian Palapunta 1086211-2202080105

Salida
- depreciación anual por año
- depreciación acumulada en cada año
- valor del automóvil en cada año

Proceso
- depreciación acumulada
- cálculo de la depreciación acumulada cada año
- cálculo del valor del automóvil en cada año

La tabla 2.1 muestra la salida solicitada.

Tabla 2.1. Salida para el problema 2.1.

Año	Depreciación	Depreciación acumulada	Valor anual
1 (2006)	3000	3000	17000
2 (2007)	3000	6000	14000
3 (2008)	3000	9000	11000
4 (2009)	3000	12000	8000
5 (2010)	3000	15000	5000
6 (2011)	3000	18000	2000

Cristian P Espana 1096211-2202050106

2.1.2. Diseño del algoritmo

En la etapa de análisis del proceso de programación se determina *qué* hace el programa; en la de diseño, *cómo lo* hace. Los métodos más eficaces para el proceso de diseño se basan en el conocido *divide y vencerás*. Es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas y a continuación dividiendo estos subproblemas en otros de nivel más bajo, hasta que pueda ser *implementada* una solución en la computadora. Este método se conoce técnicamente como **diseño descendente** (*top-down*) o **modular**. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina *refinamiento sucesivo*.

Cada subprograma se resuelve mediante un **módulo** (*subprograma*) que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un *programa principal* (el módulo de nivel más alto) que llama a subprogramas (módulos de nivel más bajo), que a su vez pueden llamar a otros subprogramas. Se dice que los programas estructurados de esta forma tienen un *diseño modular* y el método de romper el programa en módulos más pequeños se llama *programación modular*. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre sí. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar un módulo.
2. Comprobar el módulo.
3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamientos sucesivos que permitan una posterior traducción a un lenguaje se denomina **diseño del algoritmo**.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar posteriormente.

2.1.3. Herramientas de programación

Las dos herramientas más utilizadas comúnmente para diseñar algoritmos son: *diagramas de flujo* y *pseudocódigos*. Un **diagrama de flujo** (*flowchart*) es una representación gráfica de un algoritmo. Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (**ANSI**), y los más frecuentemente empleados se muestran en la figura 2.2, junto con una plantilla utilizada para el dibujo de los diagramas de flujo (figura 2.3). En la figura 2.4 se representa el diagrama de flujo que resuelve el problema 2.1.

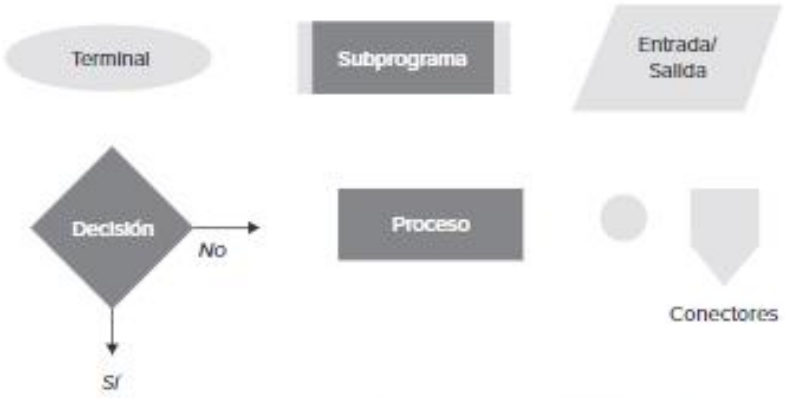


Figura 2.2. Símbolos más utilizados en los diagramas de flujo.

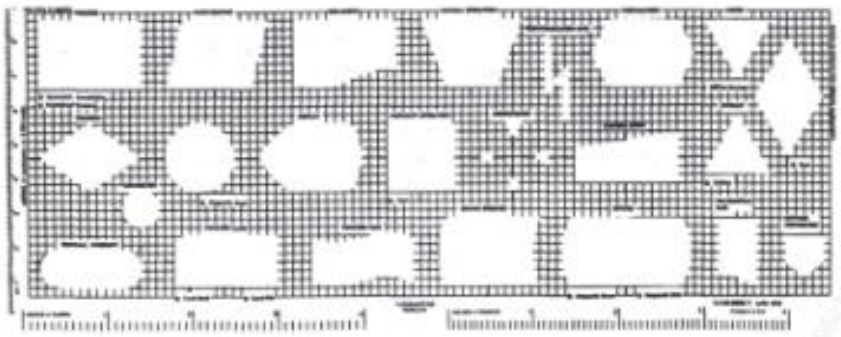


Figura 2.3. Plantilla para dibujo de diagramas de flujo.

El **pseudocódigo** (también, seudocódigo) es una herramienta de programación en la que las instrucciones se escriben en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura de programas. En esencia, el pseudocódigo se puede definir como *un lenguaje de especificaciones de algoritmos*.

Aunque no existen reglas para la escritura del pseudocódigo en español, se ha recogido una notación estándar que se utilizará en el libro y que es muy empleada en los libros de programación en español.[3] Las palabras reservadas básicas se representarán en letras negritas minúsculas. Estas palabras son traducción libre de palabras reservadas de lenguajes como C, C++, Java, Python, Pascal, etcétera. Más adelante se indicarán los pseudocódigos fundamentales para utilizar en esta obra.

El *pseudocódigo* (seudocódigo) que resuelve el problema 2.1 es:

```
Previsiones de depreciación
Introducir costo
          vida útil
          valor final de rescate (recuperación)
imprimir cabeceras
Establecer el valor inicial del año
Calcular depreciación
```

[3] Para obtener más información sobre el *pseudocódigo*, puede consultar, entre otras, algunas de estas obras: *Fundamentos de programación*, Luis Joyanes, 2a. edición, 1997; *Metodología de la programación*, Luis Joyanes, 1986; *Problemas de metodología de la programación*, Luis Joyanes, 1991 (todas ellas publicadas en McGraw-Hill, Madrid), así como *Introducción a la programación*, de Clavel y Biondi. Barcelona: Masson, 1987, o bien *Introducción a la programación y a las estructuras de datos*, de Braunstein y Groia. Buenos Aires: Editorial Eudeba, 1986. Para una formación práctica puede consultar: *Fundamentos de programación: Libro de problemas*, Luis Joyanes, Luis Rodríguez y Matilde Fernández, en McGraw-Hill (Madrid, 1998).

Cristian Pulgarín 1095211-2200092330

```
mientras valor año =< vida Útil hacer
          calcular depreciación acumulada
          calcular valor actual
          imprimir una línea en la tabla
          incrementar el valor del año
fin de mientras
```

EJEMPLO 2.1

Calcular la paga neta de un trabajador conociendo el número de horas trabajadas, la tarifa horaria y la tasa de impuestos.

Algoritmo

1. Leer Horas, Tarifa, Tasa
2. Calcular PagaBruta = Horas * Tarifa
3. Calcular Impuestos = PagaBruta * Tasa
4. Calcular PagaNeta = PagaBruta - Impuestos
5. Visualizar PagaBruta, Impuestos, PagaNeta

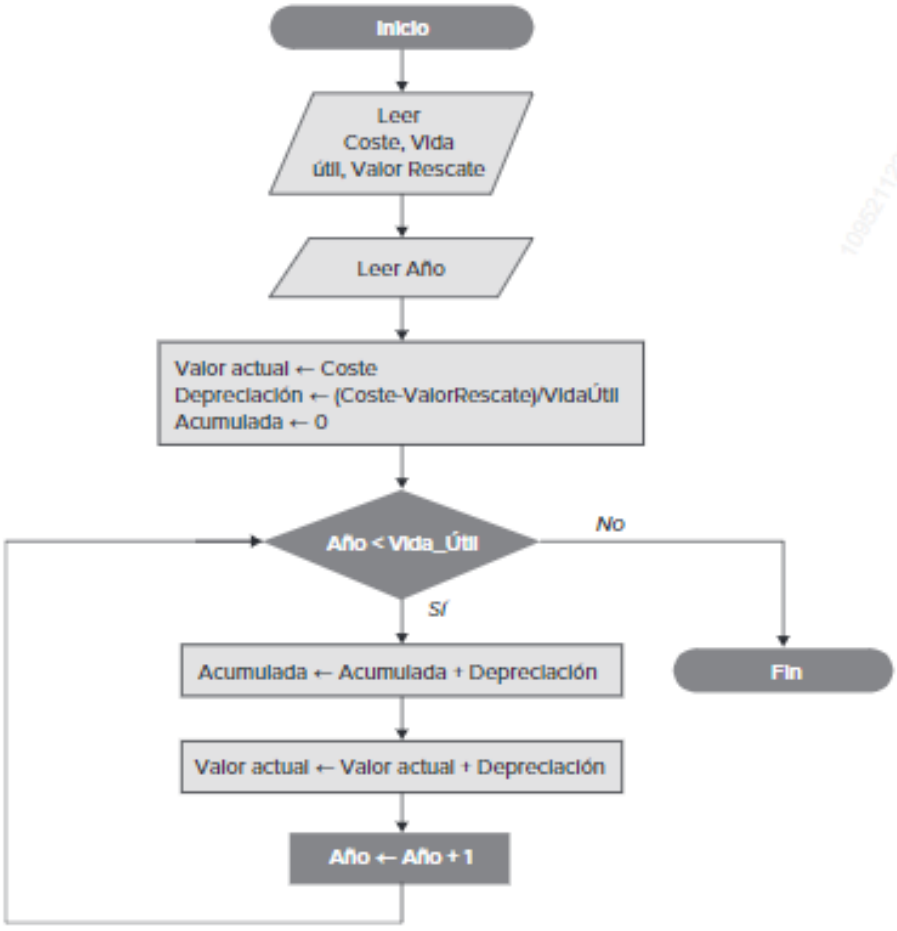


Figura 2.4. Diagrama de flujo (problema 2.1).

Cristian Pliparia 1096211-2202092330

EJEMPLO 2.2

Calcular el valor de la suma 1+2+3+...+100.

Algoritmo

Se utiliza una variable Contador como un contador que genere los sucesivos números enteros, y Suma para almacenar las sumas parciales 1, 1+2, 1+2+3...

```
1. Establecer Contador a 1
2. Establecer Suma a 0
3. mientras Contador <= 100 hacer
      Sumar Contador a Suma
      Incrementar Contador en 1
   fin_mientras
4. Visualizar Suma
```

2.1.4. Codificación de un programa

La *codificación* es la escritura, en un lenguaje de programación, de la representación del algoritmo desarrollada en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, puede ser escrito con igual facilidad en un lenguaje o en otro.

Para realizar la conversión del algoritmo a un programa, se deben sustituir las palabras reservadas en español por sus homónimos en inglés, y las operaciones/instrucciones indicadas en lenguaje natural, por el lenguaje de programación correspondiente.

```
{Este programa obtiene una tabla de depreciaciones
 acumuladas y valores reales de cada año de un
 determinado producto}
algoritmo primero
  Real: Costo, Depreciación,
        Valor_Recuperación
        Valor_Actual,
        Acumulado
        Valor_Anual;
  entero: Año, Vida_Útil;
  inicio
        escribir('introduzca costo, valor recuperación y vida útil')
        leer(Costo, Valor_Recuperación, Vida_Útil)
        escribir('Introduzca año actual')
        leer(Año) Valor_Actual ←Costo;
        Depreciación ←(Costo-Valor_Recuperación)/Vida_Útil
        Acumulado ←0
        escribir('Año Depreciación Dep. Acumulada')
        mientras (Año < Vida_Útil)
           Acumulado ←Acumulado + Depreciación
           Valor_Actual ←Valor_Actual - Depreciación
           escribir('Año, Depreciación, Acumulado')
           Año ←Año + 1;
        fin mientras
  fin
```

Cristian Pilapanta 1095211-2202092330

Documentación interna

Como se verá después, la documentación de un programa se clasifica en *interna* y *externa*. La *documentación interna* es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la comprensión del código. Todas las líneas de programas que comiencen con un símbolo / * son *comentarios*. El programa no los necesita y la computadora los ignora. Estas líneas de comentarios sólo sirven para hacer los programas más fáciles de comprender. El objetivo del programador debe ser escribir códigos sencillos y limpios.

Debido a que las máquinas actuales soportan grandes memorias (4 GB a 32 GB de memoria central mínima en computadoras personales) no es necesario recurrir a técnicas de ahorro de memoria, por lo que es recomendable que se incluya el mayor número de comentarios posibles, pero eso sí, que sean significativos.

2.1.5. Compilación y ejecución de un programa

Una vez que el algoritmo se ha convertido en un programa fuente, es preciso introducirlo en memoria mediante el teclado y almacenarlo posteriormente en un disco, pendrive (memoria usb) o en la nube. Esta operación se realiza con un programa editor. Después el programa fuente se convierte en un *archivo de programa* que se guarda (graba) en disco.

El **programa fuente** debe ser traducido a lenguaje máquina; este proceso se realiza con el compilador y el sistema operativo que se encarga prácticamente de la compilación.

Si tras la compilación se presentan errores (*errores de compilación*) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Este proceso se repite hasta que no se producen errores, obteniéndose el **programa objeto** que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de **montaje** o **enlace** (*link*), carga del programa objeto con las bibliotecas del programa del compilador. El proceso de montaje produce un **programa ejecutable**. La figura 2.5 describe el proceso completo de compilación/ejecución de un programa.

Una vez que el programa ejecutable se ha creado, ya se puede ejecutar (correr o rodar) desde el sistema operativo con sólo teclear su nombre (en el caso de DOS). Suponiendo que no existen errores durante la ejecución (llamados **errores en tiempo de ejecución**), se obtendrá la salida de resultados del programa.

Las instrucciones u órdenes para compilar y ejecutar un programa en C, C++ o cualquier otro lenguaje dependerán de su entorno de programación y del sistema operativo en que se ejecute, Windows, Linux, Unix, etcétera.

2.1.6. Verificación y depuración de un programa

La *verificación* o *compilación* de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, llamados *datos de test* o *prueba*, que determinarán si el programa tiene o no errores (*bugs*). Para realizar la verificación se debe desarrollar una amplia gama de datos de prueba: valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

La *depuración* es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores. Cuando se ejecuta un programa pueden producirse tres tipos de errores:

1. *Errores de compilación*. Se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser *errores de sintaxis*. Si existe un error de sintaxis, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.
2. *Errores de ejecución*. Estos errores se producen por instrucciones que la computadora puede comprender pero no ejecutar. Ejemplos típicos son: división por cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.
3. *Errores lógicos*. Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo puede advertirse el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.

Cristian Pilapanta 1006211-2020/02335

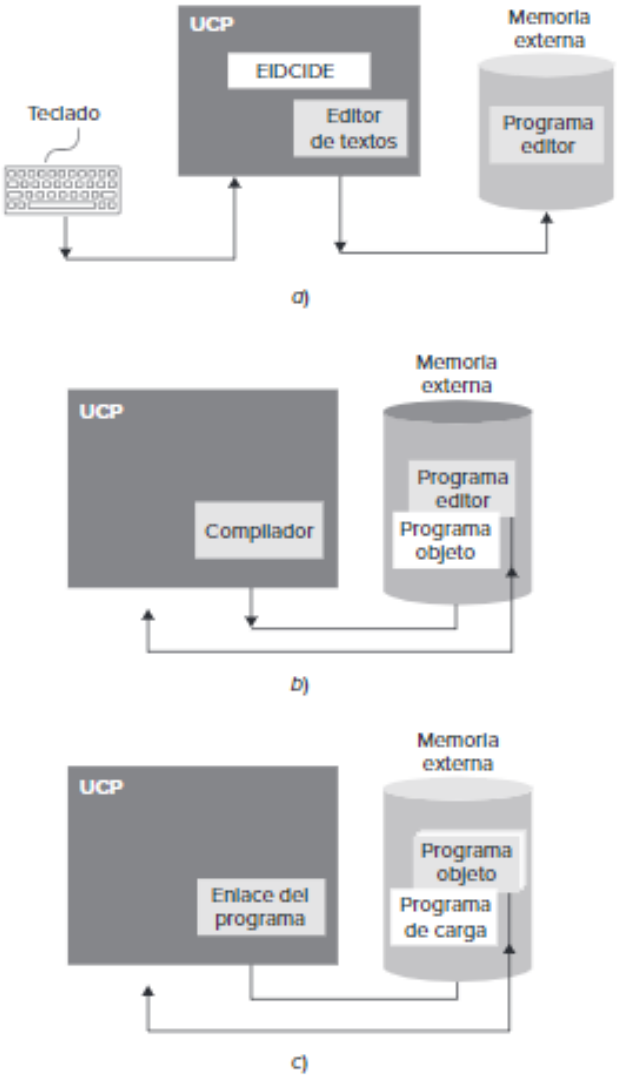


Figura 2.5. Fases de la compilación/ejecución de un programa: *a*) edición, *b*) compilación, *c*) montaje o enlace.

2.1.7. Documentación y mantenimiento

La documentación de un problema contiene las descripciones de los pasos a dar en la resolución de dicho problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas pobremente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser *interna y externa*. La *documentación interna* es la contenida en líneas de comentarios. La *documentación externa* incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Estos cambios se denominan *mantenimiento del programa*. Después de cada cambio la documentación debe ser actualizada para facilitar cambios posteriores. Es práctica frecuente numerar las sucesivas versiones de los programas **1.0, 1.1, 2.0, 2.1,** etcétera. (Si los cambios introducidos son importantes, se varía el primer dígito [**1.0, 2.0,...**]; en caso de pequeños cambios sólo se varía el segundo dígito [**2.0, 2.1...**].)

Cristian Pallqueria 1095211-2202092336

2.2. Metodología de la programación

Existen dos enfoques muy populares para el diseño y construcción de programas: enfoque estructurado y enfoque orientado a objetos. Estos conducen a dos tipos de metodologías de programación: **programación estructurada** y **programación orientada a objetos**.[4] Un tercer enfoque, la **programación modular** está directamente conectada con los dos enfoques anteriores y nació para resolver los problemas que los lenguajes primitivos imperativos presentaban y que fueron resolviéndose a medida que se realizaban los diseños por módulos (denominados en origen, *procedimientos* o *módulos*).

2.2.1. Programación modular

La **programación modular** es uno de los métodos de diseño más flexible y potente para mejorar la productividad de un programa. En programación modular el programa se divide en *módulos* (partes independientes), cada uno de los cuales ejecuta una única actividad o tarea y se codifican independientemente de otros módulos (figura 2.6). Cada uno de estos módulos se analiza, codifica y pone a punto por separado. Cada programa contiene un módulo denominado *programa principal* que controla todo lo que sucede; se transfiere el control a *submódulos* (posteriormente se denominarán *subprogramas*), de modo que estos puedan ejecutar sus funciones; sin embargo, cada submódulo devuelve el control al módulo principal cuando se haya completado su tarea. Si la tarea asignada a cada submódulo es demasiado compleja, este deberá romperse en otros módulos más pequeños. El proceso sucesivo de subdivisión de módulos continúa hasta que cada módulo tenga solamente una tarea específica que ejecutar. Esta tarea puede ser *entrada, salida, manipulación de datos, control de otros módulos* o alguna *combinación de éstos*. Un módulo puede transferir temporalmente (*bifurcar*) el control a otro módulo; sin embargo, cada módulo debe eventualmente devolver el control al módulo del cual se recibe originalmente el control.

Los módulos son independientes en el sentido en que ningún módulo puede tener acceso directo a cualquier otro módulo, excepto el módulo al que llama y sus propios submódulos. Sin embargo, los resultados producidos por un módulo pueden ser utilizados por cualquier otro módulo cuando se transfiera a ellos el control.

Cristian Pilapanta 1095211-2202092337

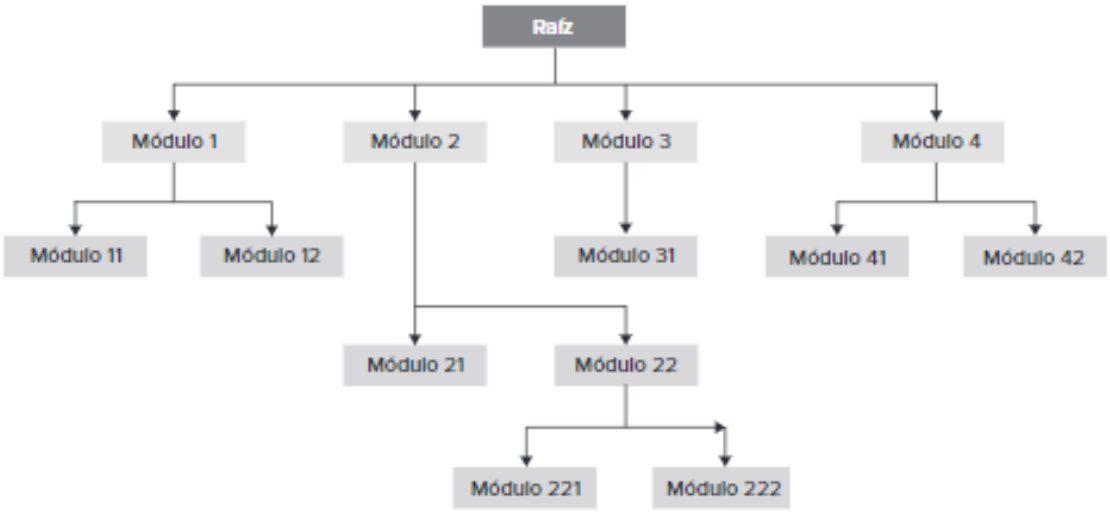


Figura 2.6. Programación modular.

[4] En la obra *Programación en C++. Un enfoque práctico*, de los profesores Luis Joyanes y Lucas Sánchez, Madrid: McGraw-Hill, 2006, puede encontrar un capítulo completo (capítulo 1) donde se analiza y comparan con detalle ambos tipos de métodos de programación.

Dado que los módulos son independientes, distintos programadores pueden trabajar simultáneamente en diferentes partes del mismo programa. Esto reducirá el tiempo del diseño del algoritmo y posterior codificación del programa. Además, un módulo se puede modificar radicalmente sin afectar a otros módulos, incluso sin alterar su función principal.

La descomposición de un programa en módulos independientes más simples se conoce también como método de *divide y vencerás* (*divide and conquer*). Cada módulo se diseña con independencia de los demás, y siguiendo un método ascendente o descendente se llegará hasta la descomposición final del problema en módulos en forma jerárquica.

La programación procedimental o procedural clásica (diseño por módulos o procedimientos) ha ido evolucionando y se consolidó como programación estructurada con la aparición de lenguajes procedimentales como C y Pascal, y posteriormente evolucionó de nuevo y se convirtió en programación orientada a objetos.

2.3. Programación estructurada

La **programación estructurada** utiliza las técnicas tradicionales del campo de programación y data de las décadas de 1960 y 1970, especialmente desde la creación del lenguaje Pascal por Niklaus Wirth. La programación estructurada es un enfoque específico que, normalmente, produce programas bien escritos y muy legibles, aunque no necesariamente un programa bien escrito y fácil de leer ha de ser estructurado. La programación estructurada trata de escribir un programa de acuerdo con unas reglas y un conjunto de técnicas.

Las reglas de programación estructurada o diseño estructurado se basan en la modularización; es decir, en la división de un problema en subproblemas más pequeños (módulos), que a su vez se pueden dividir en otros subproblemas. Cada módulo se analiza y se obtiene una solución para ese subproblema. En otras palabras, la programación estructurada implica un diseño descendente.

Una vez que se han resuelto los diferentes subproblemas o módulos, se combinan para resolver el problema global. El proceso de implementar un diseño estructurado se denomina programación estructurada. El diseño estructurado también se conoce como diseño descendente (*topdown*).

C, Pascal, Fortran y lenguajes similares se conocen como ***lenguajes procedimentales*** (*por procedimientos*). Es decir, cada sentencia o instrucción señala al compilador para que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etcétera. En resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. En el caso de pequeños programas, estos principios de organización (denominados *paradigma)* se demuestran eficientes. El programador sólo tiene que crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y esta, a su vez, ejecuta esas instrucciones.

Cuando los programas se vuelven más grandes, cosa que lógicamente sucede cuando aumenta la complejidad del problema a resolver, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Los programadores pueden controlar, de modo normal, unos centenares de líneas de instrucciones. Para resolver este problema los programas se descompusieron en unidades más pequeñas que adoptaron el nombre de *funciones* (*procedimientos*, *subprogramas* o *subrutinas* en otros lenguajes de programación). De este modo en un programa orientado a procedimientos se divide en funciones, de modo que cada función tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper el programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas ***módulos*** (normalmente, en el caso de C, denominadas **archivos** o **ficheros**); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). *Esta característica hace que a medida que los programas se hacen más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resulta muy difícil terminar los programas de un modo eficiente. Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos.* Tal vez las dos razones más evidentes son: primero, las funciones tienen acceso ilimitado a los datos globales; segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental, proporcionan un modelo pobre del mundo real.

Cristian Pilquena 1095211-2202092336

2.3.1. Datos locales y datos globales

En un programa procedimental, por ejemplo escrito en C, existen dos tipos de datos: *locales y globales*. Los *datos locales* están ocultos en el interior de la función y son utilizados exclusivamente por la función. Estos datos están estrechamente relacionados con sus funciones y están protegidos de modificaciones por otras funciones.

Los *datos globales* son aquellos a los cuales se puede acceder desde *cualquier* función del programa. Es decir, dos o más funciones pueden acceder a los mismos datos siempre que sean globales. En la figura 2.7 se muestra la disposición de variables locales y globales en un programa procedimental.

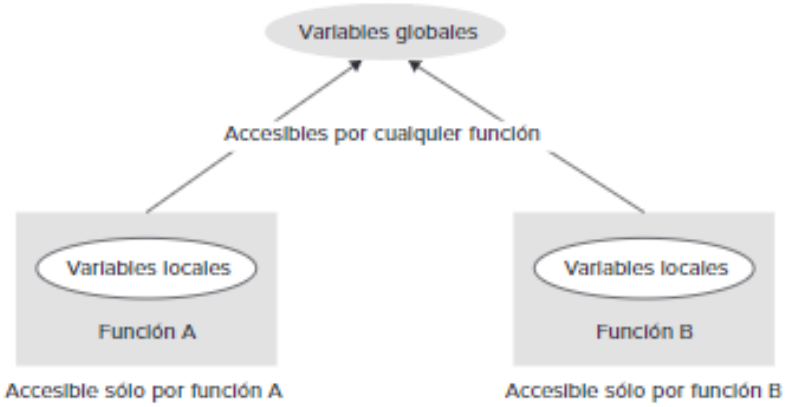


Figura 2.7. Datos locales y globales.

Un programa grande (figura 2.8) se compone de numerosas funciones y datos globales, lo que conlleva una multitud de conexiones entre funciones y datos que dificulta su comprensión y lectura.

Todas estas conexiones múltiples originan diferentes problemas. En primer lugar, hacen difícil conceptuar la estructura del programa. En segundo lugar, el programa es difícil de modificar debido a que los cambios en datos globales pueden necesitar la reescritura de todas las funciones que acceden a los mismos. También puede suceder que estas modificaciones de los datos globales puedan no ser aceptadas por todas o algunas de las funciones.

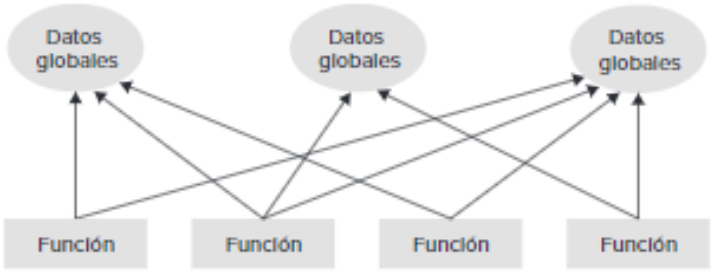


Figura 2.8. Un programa procedimental.

2.3.2. Técnicas de programación estructurada

Las técnicas de programación estructurada incluyen construcciones o estructuras (instrucciones) básicas de control.

- **Secuencia.**
- **Decisión** (también denominada *selección*).
- **Bucles** o **lazos** (también denominada *repetición* o *iteración*).

Cristian Pilapanta 1006211-2202092338

Las estructuras básicas de control especifican el orden en el que se ejecutan las distintas instrucciones de un algoritmo o programa. Una construcción (estructura, instrucción o sentencia en la jerga de lenguajes de programación) es un bloque de instrucciones de un lenguaje y una de las operaciones fundamentales del lenguaje.

Normalmente la ejecución de las sentencias o instrucciones de un programa o subprograma se realiza una después de otra en orden secuencial. Este procedimiento se llama ejecución secuencial. Existen, no obstante, diferentes sentencias que especifican cómo saltar el orden secuencial, es decir, que la sentencia a ejecutar sea distinta de la siguiente en la secuencia. Esta acción se denomina transferencia de control o control del flujo del programa. Los primeros lenguajes de programación tenían entre las sentencias de control del flujo una denominada `goto` (`ir_a`) que permitía especificar una transferencia de control a un amplio rango de destinos de un programa y poco a poco se fue abandonando por los muchos problemas que conllevaba un control eficiente (en su tiempo a este tipo de programación se denominó "código espagueti" a aquellos programas en los que se usaba esta sentencia).

En la década de 1970 nació la ya citada tendencia de programación, denominada programación estructurada que preconizaba la no utilización de la sentencia `goto` en los programas y su sustitución por otras sentencias de transferencia de control debido al gran daño que suponía a la eficiencia de los programas. Böhm y Jacopini demostraron que los programas se podían escribir sin sentencias `goto`. Sin embargo, fue también en la citada década de 1970, con la aparición de lenguajes de programación como Pascal y C en la que la tendencia se hizo una realidad y prácticamente se desechó el uso de la sentencia `goto`.

Böhm y Jacopini demostraron también que todos los programas pueden ser escritos sólo con tres estructuras de control: secuencial, de selección y de repetición. La estructura secuencial ejecuta las sentencias en el orden en que están escritas o se señala expresamente. La estructura de selección se implementa en uno de los tres formatos siguientes:

Sentencia `if (si)`: selección única
Sentencia `if-else (si-entonces-sino)`: selección doble
Sentencia `switch (según_sea)`: selección múltiple

La estructura de repetición se implementa en tres formatos diferentes:

Sentencia `while (mientras)`
Sentencia `do-while (hacer-mientras)`
Sentencia `for (desde/para)`

La *programación estructurada* promueve el uso de las tres sentencias de control:

Secuencia
Selección (sentencias, `if`, `if-else`, `switch`)
Repetición (sentencias `while`, `dowhile`, `for`)

2.3.3. Modelado del mundo real

Otro problema importante de la programación estructurada reside en el hecho de que la disposición separada de datos y funciones no se corresponden con los modelos de las cosas del mundo real. En el mundo físico se trata con objetos físicos tales como personas, autos o aviones. Estos objetos no son como los datos ni como las funciones. Los objetos complejos o no del mundo real tienen *atributos* y *comportamiento*.

Los **atributos** o características de los objetos son, por ejemplo: en las personas, su edad, su profesión, su domicilio, entre otros; en un automóvil, la potencia, el número de matrícula, el precio, número de puertas, etcétera; en una casa, la superficie, el precio, el año de construcción, la dirección, etcétera. En realidad, los atributos del mundo real tienen su equivalente en los datos de un programa; tienen un valor específico, tal como 200 m^2, 20000 dólares, cinco puertas, etcétera.

El **comportamiento** es una acción que ejecutan los objetos del mundo real como respuesta a un determinado estímulo. Si usted pisa los frenos en un auto, el coche (carro) se detiene; si acelera, el auto aumenta su velocidad, etcétera. El comportamiento, en esencia, es como una función: se llama a una función para hacer algo (visualizar la nómina de los empleados de una empresa).

Por estas razones, ni los datos ni las funciones por sí mismas, modelan los objetos del mundo real de un modo eficiente.

Cristian Pilapanta 1095211-2202092340

La programación estructurada mejora la claridad, fiabilidad y facilidad de mantenimiento de los programas; sin embargo, para programas grandes o a gran escala, presentan retos de difícil solución. Estos inconvenientes se resuelven con la programación orientada a objetos.

2.4. Programación orientada a objetos

La *programación orientada a objetos* (**POO**) tal vez el paradigma de programación más utilizado en el mundo del desarrollo de software y de la ingeniería de software del siglo XXI, trae un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación *procedimental* que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque *procedimental* de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema.

La idea fundamental de los lenguajes orientados a objetos es combinar en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama un **objeto**.

Las funciones de un objeto se llaman *funciones miembros* en C++ o *métodos* (este es el caso de Smalltalk, uno de los primeros lenguajes orientados a objetos), y son el único medio para acceder a sus datos. Los datos de un objeto, se conocen también como *atributos* o *variables de instancia*. Si se desea leer datos de un objeto, se llama a una función miembro del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente. Los datos están ocultos, de modo que están protegidos de alteraciones accidentales. Los datos y las funciones se dice que están *encapsulados en una única entidad. El encapsulamiento y la ocultación de datos* son términos clave en la descripción de lenguajes orientados a objetos.

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles son las funciones que interactúan con miembros del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa. Un programa C++ se compone normalmente de un número de objetos que se comunican unos con otros mediante la llamada a otras funciones miembro. La organización de un programa en C++ se muestra en la figura 2.9. La llamada a una función miembro de un objeto se denomina *enviar un mensaje* a otro objeto.

Cristian Pilapanta 1006211-2202002340

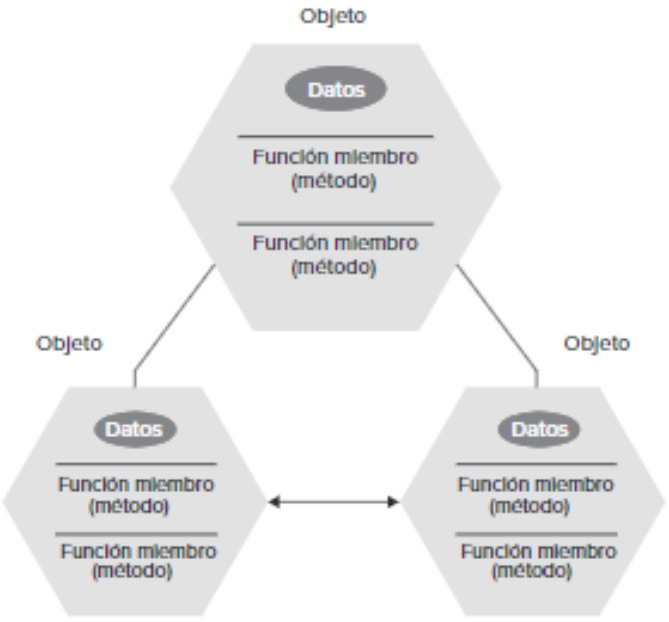


Figura 2.9. Organización típica de un programa orientado a objetos.

En el paradigma orientado a objetos, un programa se organiza como un conjunto finito de objetos que contiene datos y operaciones (*funciones miembro o métodos*) que llaman a esos datos y que se comunican entre sí mediante mensajes.

2.4.1. Abstracción

La abstracción es un elemento esencial en la programación orientada a objetos y es una propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los aspectos restantes. Los humanos gestionan la complejidad a través de la abstracción; por ejemplo, las personas cuando piensan en un automóvil (carro o coche), no lo ven como un conjunto de centenares o miles de cosas o partes individuales, sino como un objeto bien definido con su propio comportamiento. La abstracción permite a las personas conducir un coche o un carro, sin preocuparse de la complejidad de las partes individuales que lo componen. Pueden ignorar los detalles del motor, de la carrocería, del sistema de transmisión o de frenado, ven el auto como un todo y así lo utilizan.

El término **abstracción** que se suele utilizar en programación se refiere al hecho de diferenciar entre las propiedades externas de una entidad y los detalles de la composición interna de dicha entidad. Es la abstracción la que permite ignorar los detalles internos de un dispositivo complejo tal como una computadora, un automóvil, una lavadora o un horno de microondas, y usarlo como una única unidad comprensible. Mediante la abstracción se diseñan y fabrican estos sistemas complejos en primer lugar y, posteriormente, los componentes más pequeños de los cuales están compuestos. Cada componente representa un nivel de abstracción, en el cual el uso del componente se aísla de los detalles de la composición interna del componente.

En consecuencia, la abstracción posee diversos grados de complejidad que se denominan *niveles de abstracción* que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el modelado orientado a objetos de un sistema significa centrarse en *qué es* y *qué hace* un objeto y no en *cómo* debe implementarse. Durante el proceso de abstracción se decide cuáles características y comportamiento debe tener el modelo.

Al aplicar la abstracción se es capaz de construir, analizar y gestionar sistemas de computadoras complejos y grandes que no se podrían diseñar si se tratara de modelar a un nivel detallado. En cada nivel de abstracción se visualiza el sistema en términos de componentes, denominados **herramientas abstractas**, cuya composición interna se ignora. Esto nos permite concentrarnos en cómo cada componente interactúa con otros componentes y centrarnos en la parte del sistema que es más relevante para la tarea a realizar en lugar de perderse a nivel de detalles menos significativos.

En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos no sólo es de interés *cómo* están organizados, sino también *qué* se puede hacer con ellos; es decir, las operaciones que forman la interfaz de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (TAD). Un tipo abstracto de datos describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto.

Las abstracciones jerárquicas de sistemas complejos también se aplican a los programas de computadora. Los datos de un programa tradicional orientado a procesos se pueden transformar por abstracción a objetos componentes. Una secuencia de etapas de proceso se puede convertir en una colección de mensajes entre estos objetos. Por consiguiente, cada uno de estos objetos describe su propio comportamiento único. Es posible tratar estos objetos como entidades concretas que responden a mensajes que les indican que han de hacer algo. Esta es la esencia de la programación orientada a objetos.

Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

EJEMPLO 2.3

Diferentes modelos de abstracción del término coche (carro).

- Un `coche (carro)` es la combinación (o composición) de diferentes partes, tales como motor, carrocería, cuatro ruedas, cinco puertas, etcétera.

Cristian Pilapanta 1006211-2202092341

- Un `coche (carro)` es un concepto común para diferentes tipos de coches. Pueden clasificarse por el nombre del fabricante (Audi, BMW, Mercedes, Seat, Toyota, Chrysler...), por su categoría (turismo, deportivo, todoterreno...), por el carburante que utilizan (gasolina, gasoil, gas, híbrido...).

La abstracción `coche` se utilizará siempre que la marca, la categoría o el carburante no sean significativos. Así, un `carro (coche)` se utilizará para transportar personas o ir de Carchelejo a Cazorla.

Además de la característica fundamental de la *abstracción*, los tres principios básicos por los que se rige la programación orientada a objetos son: *encapsulamiento*, *herencia* y *polimorfismo*; a continuación, haremos una breve introducción a dichos conceptos.

Existen diversas características ligadas a la orientación a objetos. Todas las propiedades que se suelen considerar no son exclusivas de este paradigma, ya que pueden existir en otros paradigmas, pero en su conjunto definen claramente los lenguajes orientados a objetos. Los **principios fundamentales de la orientación a objetos**, además de la característica antes descrita de abstracción son:

- **Encapsulamiento.**
- **Herencia.**
- **Polimorfismo.**

Cristian Pilapanta 1006211-2202092342

Encapsulamiento

El **encapsulamiento** (o *encapsulado*) de datos es el mecanismo que enlaza el código y los datos que manipula y mantiene a ambas entidades seguras de interferencias externas y mal uso. En esencia, es como si se introdujera en una cápsula protectora que evita que el código y los datos se puedan acceder de modo arbitrario por otros códigos definidos fuera de la envoltura. El acceso al código y los datos dentro de la envoltura se controlan estrictamente a través de una interfaz bien definida.

En C++, Python y en Java, la unidad de encapsulamiento es la clase. Una **clase** define la estructura y comportamiento (datos y código) que se comparte por un conjunto de objetos. Cada objeto de una clase especificada contiene la estructura y el comportamiento definido por la clase como si fuera estampado por un molde en el formato de la clase. Por esta razón los objetos se conocen como *instancias* de una clase. En consecuencia, una clase es una construcción lógica y un objeto tiene una realidad física.

Cuando se crea una clase, se debe especificar el código y los datos que constituyen dicha clase. De modo colectivo estos elementos se llaman *miembros* de la clase. Específicamente, los datos definidos en la clase se conocen como *miembros variable* o *variables de instancia*. El código que opera sobre los datos se conoce como miembros método o, simplemente, *métodos* en C++ o *funciones* en Java. En programas Java escritos correctamente los métodos definen cómo se pueden utilizar las variables miembros, lo que significa que el comportamiento y la interfaz de una clase se definen por los métodos que operan sobre sus datos de instancia.

Dado que el propósito de una clase es encapsular la complejidad, existen mecanismos para ocultar la complejidad de la implementación dentro de la clase. Cada método o variable de una clase se pueden definir como privado o público. La interfaz pública de una clase representa todo lo que los usuarios externos de la clase necesitan conocer o pueden conocer. Los métodos y datos privados sólo se pueden acceder por el código que es miembro de la clase. Por consiguiente, cualquier otro código que no sea un miembro de la clase no puede acceder a un método o variable privada. Dado que a los miembros privados de una clase sólo se puede acceder por otras partes de su programa a través de métodos públicos de la clase, se puede asegurar que no se podrá realizar ninguna acción inapropiada. Eso entraña que la interfaz pública se debe diseñar cuidadosamente para no exponer demasiado el funcionamiento interno de una clase.

La ocultación de datos permite separar el aspecto de un componente, definido por su *interfaz* con el exterior, de sus detalles internos de implementación. Los términos **ocultación de la información** (*information hiding*) y **encapsulación de datos** (*data encapsulation*) suelen utilizarse como sinónimos, pero no siempre es así y, muy al contrario,

son términos similares, pero distintos. Normalmente, los datos internos están protegidos del exterior y no se puede acceder a ellos más que desde su propio interior y, por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.

El **encapsulamiento** (*encapsulado* o *encapsulación*) **de datos** es, en resumen, el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de los objetos que poseen las mismas características y comportamiento se agrupan en clases, que no son más que unidades o módulos de programación que encapsulan datos y operaciones.

Herencia

Es el proceso por el que un objeto adquiere las propiedades de otro. Esta propiedad es muy importante debido a que soporta el concepto de clasificación jerárquica. El conocimiento humano se realiza y gestiona de modo jerárquico. Por ejemplo, un perro pastor alemán es parte de una clasificación de la clase mamífero que a su vez es miembro de la clase animal. Sin el uso de jerarquías cada objeto necesitará definir sólo aquellas cualidades específicas o únicas dentro de su clase. Puede heredar atributos generales de sus padres. Por consiguiente, el mecanismo de la herencia hace posible que un objeto sea una instancia específica de un caso más general.

Si desea describir una clase más específica de animales, tendrán atributos específicos como el tipo de dientes o las glándulas mamarias, o los tipos de patas o piernas. Este caso se conoce como una **subclase** de animales que a su vez es una **superclase** de mamífero.

La herencia interactúa con el encapsulamiento. Si una clase dada encapsula algunos atributos, entonces cualquier subclase tendrá los mismos atributos más cualquier otro que se añada como parte de la especialización. Este es un concepto clave que permite a los programas crecer linealmente en complejidad en lugar de geométricamente.

> Una nueva subclase hereda todos los atributos de todos sus ascendientes

El principio de herencia se basa en el concepto de *generalización*, propiedad que permite compartir información entre dos entidades evitando la redundancia. En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos, esta propiedad se denomina generalización.

Por ejemplo, máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas, etcétera, son todos electrodomésticos (aparatos del hogar). En el mundo de la orientación a objetos, cada uno de estos aparatos es una `subclase` de la clase `Electrodoméstico` y a su vez `Electrodoméstico` es una `superclase` de todas las otras clases (`máquinas lavadoras`, `frigoríficos`, `hornos de microondas`, `tostadoras`, `lavavajillas`). El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina *especialización*.

En orientación a objetos, el mecanismo que implementa la propiedad de generalización se denomina `herencia`. La herencia permite definir nuevas clases a partir de otras ya existentes, de modo que presentan las mismas características y comportamiento de éstas, así como otras adicionales. En esencia, herencia es el proceso por el que un objeto adquiere las propiedades de otro objeto; es decir, se apoya en el concepto de clasificación jerárquica al estilo del conocimiento humano que se realiza y gestiona de modo jerárquico; por ejemplo, un `perro pastor alemán`, pertenece a la clase `mamífero` que a su vez es miembro de la clase `animal`.

La idea de clases conduce a la idea de **herencia**. Clases diferentes se pueden conectar unas con otras de modo jerárquico. Como ya se ha comentado anteriormente con las relaciones de generalización y especialización, en nuestras vidas diarias se utiliza el concepto de clases divididas en subclases. La clase `animal` se divide en `anfibios`, `mamíferos`, `insectos`, `pájaros`, etcétera, y la clase `vehículo` en `carros`, `motos`, `camiones`, `buses`, entre otros.

El principio de la división o clasificación es que cada subclase comparte características comunes con la clase de la que procede o se deriva. Los carros, motos, camiones y buses tienen ruedas, motores y carrocerías; son las características que definen a un vehículo. Además de las características comunes con los otros miembros de la clase, cada subclase tiene sus propias características. Por ejemplo, los camiones tienen una cabina independiente de la caja que transporta la carga; los buses tienen un gran número de asientos independientes para los viajeros que ha de transportar, etcétera. En la figura 2.10 se muestran clases pertenecientes a una jerarquía o **herencia de clases**.

Cristian Pilapanta 1096211-2202092344

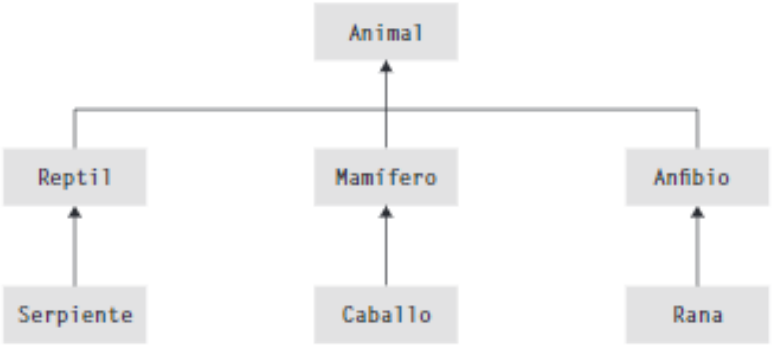


Figura 2.10. Herencia de clases en UML.

De modo similar una clase se puede convertir en padre o raíz de otras subclases. En C++, la clase original se denomina *clase base* y las clases que se derivan de ella se denominan *clases derivadas* y siempre son una especialización o *concreción* de su clase base. A la inversa, la clase base es la generalización de la clase derivada. Esto significa que todas las propiedades (atributos y operaciones) de la clase base se heredan por la clase derivada, normalmente suplementada con propiedades adicionales.

Cristian Pilapanta 1006211-2202002344

Polimorfismo

El término **polimorfismo** significa múltiples formas. Es una característica que permite que una interfaz se pueda utilizar para una clase general de acciones. La acción específica se determina por la naturaleza exacta de la situación. Un ejemplo clásico se puede ver en una pila como una estructura de datos (véase el capítulo 7) "última en entrar, primera en salir, UEPS". Se puede necesitar un programa que requiera tres tipos de pilas. Una pila se utiliza para valores enteros, otra para valores de coma flotantes y otra para datos tipo carácter. El algoritmo que implementa cada pila es el mismo, aunque el sistema de almacenamiento de los datos es diferente.

En un lenguaje no orientado a objetos se requerirá crear tres diferentes conjuntos de rutinas o funciones de pila con cada conjunto utilizando un nombre diferente. Sin embargo, mediante el polimorfismo en Java o en C++ se puede especificar un conjunto general de funciones o rutinas de pila que comparten el mismo nombre.

Además de las ventajas de consistencia y reducción de código, la herencia aporta también otra gran ventaja: facilitar el polimorfismo. Polimorfismo, como se mencionó anteriormente, es la propiedad de que un operador o una función actúen de modo diferente en función del objeto sobre el que se aplican. En la práctica, el polimorfismo significa la capacidad de una operación de ser interpretada sólo por el propio objeto que lo invoca. Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución ya que durante la compilación no se conoce qué tipo de objeto y, por consiguiente, cuál operación ha sido llamada.

La propiedad de **polimorfismo** es aquella en la que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. Así, por ejemplo, la operación de `abrir` se puede dar en diferentes clases: abrir una puerta, abrir una ventana, abrir un periódico, abrir un archivo, abrir una cuenta corriente en un banco, abrir un libro, etcétera. En cada caso se ejecuta una operación diferente aunque tiene el mismo nombre en todos ellos "abrir". Existen diferentes formas de implementar el polimorfismo y variará dependiendo del lenguaje de programación. Veamos el concepto con ejemplos de la vida diaria.

En un taller de reparaciones de automóviles existen numerosos carros, de marcas diferentes, de modelos diferentes, de tipos diferentes, potencias diferentes, etcétera. Constituyen una clase o colección heterogénea de carros (coches). Supongamos que se ha de realizar una operación común como "cambiar los frenos del carro". La operación a realizar es la misma, incluye los mismos principios; sin embargo, dependiendo del coche en particular, la operación será muy distinta, incluirá diferentes acciones en cada caso. Otro ejemplo a considerar y relativo a los operadores "+" y "*" aplicados a números enteros o números complejos; aunque ambos son números, en un caso la suma y multiplicación son operaciones simples, mientras que en el caso de los números complejos al componerse de parte real y parte imaginaria, será necesario seguir un método específico para tratar ambas partes y obtener un resultado que también será un número complejo.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que están actuando se llama polimorfismo (una cosa con diferentes formas). Sin embargo, cuando un operador existente, tal como + o =, se le permite la posibilidad de operar sobre nuevos tipos de datos, se dice entonces que el operador está sobrecargado. La sobrecarga es un tipo de polimorfismo y una característica importante de la POO.

El concepto de polimorfismo de modo más general se expresa en el término "una interfaz, múltiples métodos". Esto significa que es posible diseñar una interfaz genérica para un grupo de actividades relacionadas. Esta característica ayuda a reducir la complejidad permitiendo que la misma interfaz se utiliza para especificar una clase general de acciones. Es tarea del compilador seleccionar la acción específica (esto es un método) que se aplica a cada situación. El programador no necesita realizar la selección manualmente. Sólo se necesita recordar y utilizar la interfaz general. El concepto general se puede implementar en Java a medida que se aplica a métodos dentro de un programa Java.

Los tres principios anteriores se combinan para producir un entorno de programación que soporta el desarrollo de programas más robustos y escalables que el modelo procedimental.

- Una jerarquía de clases bien diseñada es el fundamento para reutilizar el código en el que se ha invertido tiempo y esfuerzo para desarrollar y probar.
- El encapsulamiento permite migrar sus implementaciones en el tiempo sin romper el código que depende de la interfaz pública de sus clases.
- El polimorfismo le permite crear código limpio, sensible, legible y resiliente.

Cada programa implica encapsulamiento, herencia y polimorfismo, las librerías de clases integradas en Java hacen un uso extensivo de los principios fundamentales.

Cristian Pilapanta 1096211-2202002345

2.5. Concepto y características de algoritmos

El objetivo fundamental de este texto es enseñar a resolver problemas mediante una computadora. El programador de computadora es una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A lo largo de todo este libro nos referiremos a la *metodología necesaria para resolver problemas mediante programas*, concepto que se denomina **metodología de la programación**. El eje central de esta metodología es el concepto, ya tratado, de algoritmo.

Un algoritmo es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, **algoritmo** proviene —como se comentó anteriormente— de *Mohammed alKhoWârizmi*, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra *algorismus* derivó posteriormente en algoritmo. Euclides, el gran matemático griego del siglo IV a. C. que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al-Khowârizmi el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth —inventor de Pascal, Modula-2 y Oberon— tituló uno de sus más famosos libros *Algoritmos + Estructuras de datos = Programas*, significándonos que sólo se puede realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas en esta obra.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto (figura 2.11).

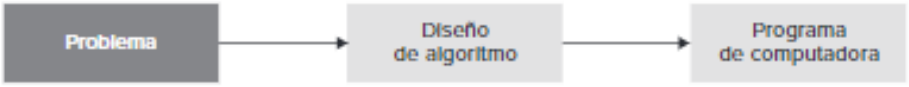


Figura 2.11. Resolución de un problema.

Los pasos para la resolución de un problema son:

1. *Diseño del algoritmo*, que describe la secuencia ordenada de pasos —sin ambigüedades— que conducen a la solución de un problema dado. (*Análisis del problema y desarrollo del algoritmo.*)

2. Expresar el algoritmo como un *programa* en un lenguaje de programación adecuado. (*Fase de codificación*).
3. *Ejecución y validación* del programa por la computadora.

Para la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema, el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será *el diseño de algoritmos*.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, *la solución de un problema se puede expresar mediante un algoritmo*.

Cristian Palapanta 1095211-2200092346

2.5.1. Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Debe ser *preciso* e indicar el orden de realización de cada paso.
- Debe estar bien *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada*, *Proceso* y *Salida*. En el algoritmo de receta de cocina citado anteriormente se tendrá:

Entrada: Ingredientes y utensilios empleados.
Proceso: Elaboración de la receta en la cocina.
Salida: Terminación del plato (por ejemplo, cordero).

EJEMPLO 2.4

Un cliente realiza un pedido a una fábrica. La fábrica examina en su banco de datos la ficha del cliente, si el cliente es solvente entonces la empresa acepta el pedido; caso contrario, rechazará el pedido. Redactar el algoritmo correspondiente.

Los pasos del algoritmo son:

1. `Inicio.`
2. `Leer el pedido.`
3. `Examinar la ficha del cliente.`
4. `Si el cliente es solvente, aceptar pedido; caso contrario, rechazar pedido.`
5. `Fin.`

EJEMPLO 2.5

Se desea diseñar un algoritmo para saber si un número es primo o no.

Un número es primo sólo si puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6, 4, 12, 16, 20, etcétera, no son primos, ya que son divisibles por números

distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etcétera. El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4..., etcétera.

1. Inicio.
2. Poner X igual a 2 (X ← 2, X variable que representa a los divisores del número que se busca N).
3. Dividir N por X (N/X).
4. Si el resultado de N/X es entero, entonces N no es un número primo y bifurcar al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a X (X ← X + 1).
6. Si X es igual a N, entonces N es un número primo; en caso contrario, bifurcar al punto 3.
7. Fin.

Por ejemplo, si N es 131, los pasos anteriores serían:

1. Inicio.
2. X = 2.
3. 131/X. Como el resultado no es entero, se continúa el proceso. 5. X ← 2 + 1, luego X = 3.
4. Como X no es 131, se continúa el proceso.
5. 131/X resultado no es entero. 5. X ← 3 + 1, X = 4.
6. Como X no es 131 se continúa el proceso. 3. 131/X..., etc.
7. Fin.

EJEMPLO 2.6

Realizar la suma de todos los números pares entre 2 y 1000.

El problema consiste en sumar 2 + 4 + 6 + 8 ... + 1000. Utilizaremos las palabras SUMA y NÚMERO (*variables*, serán denominadas más tarde) para representar las sumas sucesivas (2+4), (2+4+6), (2+4+6+8), etcétera. La solución se puede escribir con el siguiente algoritmo:

1. Inicio.
2. Establecer SUMA a 0.
3. Establecer NÚMERO a 2.
4. Sumar NÚMERO a SUMA. El resultado será el nuevo valor de la suma (SUMA).
5. Incrementar NÚMERO en 2 unidades.
6. Si NÚMERO =< 1 000 bifurcar al paso 4; caso contrario, escribir el último valor de SUMA y terminar el proceso.
7. Fin.

2.5.2. Diseño del algoritmo

Una computadora solo tiene capacidad para solucionar problemas cuando se le proporcionan los pasos sucesivos a realizar. Estos pasos sucesivos que indican las instrucciones a ejecutar por la máquina constituyen, como ya sabemos, el *algoritmo*.

La información proporcionada al algoritmo constituye su *entrada* y la información producida por este constituye su *salida*.

Los problemas complejos se pueden resolver de manera más eficaz con la computadora cuando se dividen en subproblemas que sean más fáciles de solucionar que el original. Es el método de *divide y vencerás*, mencionado anteriormente, y que consiste en dividir un problema complejo en otros más simples. Así, el problema de encontrar la superficie y la longitud de un círculo se puede dividir en tres problemas más simples o *subproblemas* (figura 2.12).

La descomposición del problema original en subproblemas más simples y a continuación la división de estos subproblemas en otros más simples que pueden ser implementados para su solución en la computadora, se denomina *diseño descendente* (*top-down design*). Normalmente, los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán sólo unos pocos pasos (un máximo de doce aproximadamente). Tras esta primera descrip-

Cristian Pilapanta 1095211-2202092347

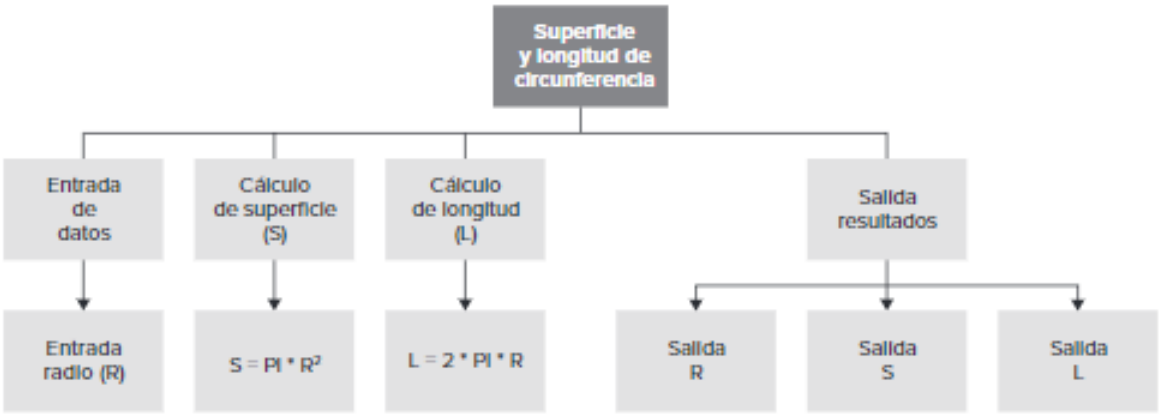


Figura 2.12. Refinamiento de un algoritmo.

ción, estos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina *refinamiento del algoritmo* (*stepwise refinement*). Para problemas complejos se necesitan con frecuencia diferentes *niveles de refinamiento* antes de que se pueda obtener un algoritmo claro, preciso y completo.

El problema de cálculo de la circunferencia y superficie de un círculo se puede descomponer en subproblemas más simples: 1) leer datos de entrada; 2) calcular superficie y longitud de circunferencia, y 3) escribir resultados (datos de salida).

Subproblema	Refinamiento
leer radio	leer radio
calcular superficie	superficie ← 3.141592 * radio ^ 2
calcular circunferencia	circunferencia ← 2 * 3.141592 * radio
escribir resultados	*escribir* radio, circunferencia, superficie

Las *ventajas* más importantes del diseño descendente son:

- El problema se comprende más fácilmente al dividirse en partes más simples denominadas *módulos*.
- Las modificaciones en los módulos son más fáciles.
- La comprobación del problema se puede verificar fácilmente.

Tras los pasos anteriores (*diseño descendente y refinamiento por pasos*) es preciso representar el algoritmo mediante una determinada herramienta de programación: *diagrama de flujo, pseudocódigo* o *diagrama N-S*.

Así pues, el diseño del algoritmo se descompone en las fases recogidas en la figura 2.13.

Figura 2.13. Fases del diseño de un algoritmo.

Cristian Pilapanta 1095211-2202002348

2.6. Escritura de algoritmos

Como ya se ha comentado anteriormente, el sistema para describir ("escribir") un algoritmo consiste en realizar una descripción paso a paso con un lenguaje natural del citado algoritmo. Recordemos que un algoritmo es un método o conjunto de reglas para solucionar un problema. En cálculos elementales estas reglas tienen las siguientes propiedades:

- Deben ir seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente.
- Solo puede ejecutarse una operación a la vez.

El flujo de control común de un algoritmo es secuencial; consideremos el algoritmo que responde a la pregunta:

¿Qué hacer para ver la película de Harry Potter?

La respuesta es muy sencilla y puede ser descrita en forma de algoritmo general de modo similar a:

```
ir al cine
comprar una entrada (billete o ticket)
ver la película
regresar a casa
```

El algoritmo consta de cuatro acciones básicas, cada una de las cuales debe ser ejecutada antes de realizar la siguiente. En términos de computadora, cada acción se codificará en una o varias sentencias que ejecutan una tarea particular.

El algoritmo descrito es muy sencillo; sin embargo, como ya se ha indicado en párrafos anteriores, el algoritmo general se descompondrá en pasos más simples en un procedimiento denominado *refinamiento sucesivo*, ya que cada acción puede descomponerse a su vez en otras acciones simples. Así, por ejemplo, un primer refinamiento del algoritmo ir al cine se puede describir de la forma siguiente:

```
1. inicio
2. ver la cartelera de cines en el periódico
3. si no proyectan "Harry Potter" entonces
   3.1 decidir otra actividad
   3.2 bifurcar al paso 7
   si_no
   3.3 ir al cine
   fin_si
4. si hay cola entonces
   4.1 ponerse en ella
   4.2 mientras haya personas delante hacer
       4.2.1 avanzar en la cola
       fin_mientras
   fin_si
5. si hay localidades entonces
   5.1 comprar una entrada
   5.2 pasar a la sala
   5.3 localizar la(s) butaca(s)
   5.4 mientras proyectan la película hacer
       5.4.1  ver la película
       fin_mientras
   5.5 abandonar el cine
   si_no
   refunfuñar
   fin_si
6. volver a casa
7. fin
```

Cristian P Hispania 1096211-2202100301

En el algoritmo anterior existen diferentes aspectos a considerar. En primer lugar, ciertas palabras reservadas se han escrito deliberadamente en negrita (**mientras**, **si_no**; etcétera). Estas palabras describen las estructuras de control fundamentales y procesos de toma de decisión en el algoritmo. Estas incluyen los conceptos importantes de *selección* (expresadas por **si-entonces-si_no**, ***if-then-else***) y de *repetición* (expresadas con **mientras-hacer** o a veces **repetir-hasta** e **iterar-fin_iterar**, en inglés, ***while-do*** y ***repeat-until***) que se encuentran en casi todos los algoritmos, especialmente en los de proceso de datos. La capacidad de decisión permite seleccionar alternativas de acciones a seguir o bien la repetición una y otra vez de operaciones básicas.

```
si proyectan la película seleccionada ir al cine
   si_no ver la televisión, ir al fútbol o leer el periódico
mientras haya personas en la cola, ir avanzando repetidamente
   hasta llegar a la taquilla
```

Otro aspecto a considerar es el método elegido para describir los algoritmos: empleo de *indentación* (sangrado o justificación) en escritura de algoritmos. En la actualidad es tan importante la escritura de programa como su posterior lectura. Ello se facilita con la *indentación* de las acciones interiores a las estructuras fundamentales citadas: selectivas y repetitivas. A lo largo de todo el libro la indentación o sangrado de los algoritmos será norma constante.

Para terminar estas consideraciones iniciales sobre algoritmos, describiremos las acciones necesarias para refinar el algoritmo objeto de nuestro estudio; para ello analicemos la acción:

```
Localizar la(s) butaca(s).
```

Si los números de los asientos están impresos en la entrada, la acción compuesta se resuelve con el siguiente algoritmo:

```
1. inicio //algoritmo para encontrar la butaca del espectador
2. caminar hasta llegar a la primera fila de butacas
3. repetir
      compara número de fila con número impreso en billete
      si son iguales entonces pasar a la siguiente fila fin_si
   hasta_que se localice la fila correcta
4. mientras número de butaca no coincida con número de billete
      hacer avanzar a través de la fila a la siguiente butaca
   fin_mientras
5. sentarse en la butaca
6. fin
```

En este algoritmo la repetición se ha mostrado de dos modos utilizando ambas notaciones, **repetir... hasta_que** y **mientras... fin_mientras**. Se ha considerado también, como ocurre normalmente, que el número del asiento y fila coincide con el número y fila rotulado en el billete.

2.7. Representación gráfica de los algoritmos

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitirá que un algoritmo pueda ser codificado indistintamente en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que las acciones sucesivas no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción pueda servir fácilmente para su transformación en un programa, es decir, *su codificación*.

Los métodos usuales para representar un algoritmo son:

1. *Diagrama de flujo.*
2. *Diagrama N-S (Nassi-Schneiderman).*
3. *Lenguaje de especificación de algoritmos: pseudocódigo.*
4. *Lenguaje español, inglés...*
5. *Fórmulas.*

Cristian Pilapanta 1095211-2202100002

Los métodos 4 y 5 no suelen ser fáciles de transformar en programas. Una descripción en *español narrativo* no es satisfactoria, ya que es demasiado prolija y generalmente ambigua. Una *fórmula*, sin embargo, es un buen sistema de representación. Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado) son un medio sucinto de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

$$x1 = (-b + \sqrt{b^2 - 4ac})/2a \qquad x2 = (-b - \sqrt{b^2 - 4ac})/2a$$

y significa lo siguiente:

1. *Elevar al cuadrado b.*
2. *Tomar a, multiplicar por c, multiplicar por 4.*
3. *Restar el resultado obtenido de 2 del resultado de 1, etcétera.*

Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

2.7.1. Pseudocódigo

El **pseudocódigo** (*seudocódigo*) es *un lenguaje de especificación (descripción) de algoritmos*. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil. Los lenguajes APL, Pascal y Ada se utilizan a veces como lenguajes de especificación de algoritmos.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada que se verán en capítulos posteriores. Se considera un *primer borrador*, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La *ventaja del pseudocódigo* es que en su uso en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que está codificado en un lenguaje de programación. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a lenguajes estructurados como Pascal, C, Fortran 77/90, C++, Java, C#, etcétera.

El pseudocódigo original utiliza palabras reservadas en inglés —similares a sus homónimas en los lenguajes de programación— para representar las acciones sucesivas, tales como `start`, `end`, `stop`, `if-then-else`, `while-end`, `repeat-until`, etcétera. La escritura de pseudocódigo exige normalmente la *indentación* (sangría en el margen izquierdo) de diferentes líneas.

Una representación en pseudocódigo —en inglés— de un problema de cálculo del salario neto de un trabajador es la siguiente:

```
start
   //cálculo de impuesto y salarios
   read nombre, horas, precio
   salario ← horas * precio
   tasas   ← 0,25 * salario
   salario_neto ← salario - tasas
   write nombre, salario, tasas, salario
end
```

El algoritmo comienza con la palabra `start` y finaliza con la palabra `end`, en inglés (en español, `inicio`, `fin`). Entre estas palabras sólo se escribe una instrucción o acción por línea.

La línea precedida por `//` se denomina *comentario*. Es una información al lector del programa y no realiza ninguna instrucción ejecutable, solo tiene efecto de documentación interna del programa. Algunos autores suelen utilizar corchetes o llaves.

No es recomendable el uso de apóstrofos o simples comillas como representan en algunos lenguajes primitivos los comentarios, ya que este carácter es representativo de apertura o cierre de cadenas de caracteres en lenguajes como Pascal o Fortran y daría lugar a confusión.

Cristian Plaganis 106211-2202100003

Otro ejemplo aclaratorio en el uso del pseudocódigo podría ser un sencillo algoritmo del arranque matinal de un coche.

```
inicio
   //arranque matinal de un coche
   introducir la llave de contacto
   girar la llave de contacto
   pisar el acelerador
   oír el ruido del motor
   pisar de nuevo el acelerador
   esperar unos instantes a que se caliente el motor
fin
```

Por fortuna, aunque el pseudocódigo nació como un sustituto del lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las del idioma inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como **inicio**, **fin**, **parada**, **leer**, **escribir**, **si-entonces-si_no**, **mientras**, **fin_mientras**, **repetir**, **hasta_que**, etcétera. Sin duda, el uso de la terminología del pseudocódigo en español ha facilitado y facilitará considerablemente el aprendizaje y uso diario de la programación. En esta obra, al igual que en otras del mismo autor, utilizaremos el pseudocódigo en español y daremos en su momento las estructuras equivalentes en inglés, al objeto de facilitar la traducción del pseudocódigo al lenguaje de programación seleccionado.

Así pues, en los pseudocódigos citados anteriormente deberían ser sustituidas las palabras **start**, **end**, **read**, **write**, por **inicio**, **fin**, **leer**, **escribir**, respectivamente.

inicio	*start*	*leer*	*read*
.			
.			
.			
.			
.			
fin	*end*	*escribir*	*write*

Cristian Palpanta 1085211-2202100204

2.7.2. Diagramas de flujo

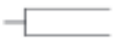
Un **diagrama de flujo** (*flowchart*) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar mostrados en la tabla 2.2 y que tiene los pasos de algoritmo escritos en esas cajas unidas por flechas, denominadas *líneas de flujo*, que indican la secuencia en que se debe ejecutar.

La figura 2.14 es un diagrama de flujo básico. Este diagrama representa la resolución de un programa que deduce el salario neto de un trabajador a partir de la lectura del nombre, horas trabajadas, precio de la hora, y sabiendo que los impuestos aplicados son 25% sobre el salario bruto.

Los símbolos estándar normalizados por **ANSI** (abreviatura de *American National Standards Institute*) son muy variados. En la figura 2.15 se representa una plantilla de dibujo típica donde se contemplan la mayoría de los símbolos utilizados en el diagrama; sin embargo, los símbolos más utilizados representan:

- proceso
- decisión
- conectores
- fin
- entrada/salida
- dirección del flujo

Tabla 2.2. Símbolos de diagrama de flujo.

Símbolos principales	Función
	Terminal (representa el comienzo, "inicio", y el final, "fin" de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa.
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, "entrada", o registro de la información procesada en un periférico, "salida".
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etcétera).
	Decisión (indica operaciones lógicas o de comparación entre datos —normalmente dos— y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas —respuestas SÍ o NO— pero puede tener tres o más, según los casos).
	Decisión múltiple (en función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector sirve para enlazar dos partes cualesquiera de un ordinograma a través de un conector en la salida y otro conector en la entrada. Se refiere a la conexión en la misma página del diagrama.
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector (conexión entre dos puntos del organigrama situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independientemente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).

El diagrama de flujo de la figura 2.14 resume sus características:

- Existe una caja etiquetada "`inicio`", que es de tipo elíptico.
- Existe una caja etiquetada "`fin`" de igual forma que la anterior.
- Si existen otras cajas, normalmente son rectangulares, tipo rombo o paralelogramo (el resto de las figuras se utilizan sólo en diagramas de flujo generales o de detalle y no siempre son imprescindibles).

Cristian Palpanta 1006211-2202100205

Problema:
Calcular el salario bruto y el salario neto de un trabajador "por horas" conociendo el nombre, número de horas trabajadas, impuestos a pagar y salario neto.

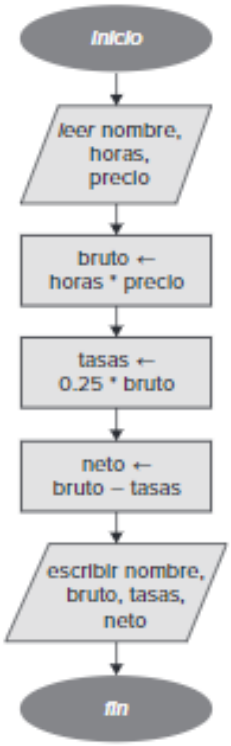


Figura 2.14. Diagrama de flujo.

Cristian Palapanta 1065211-2202100305

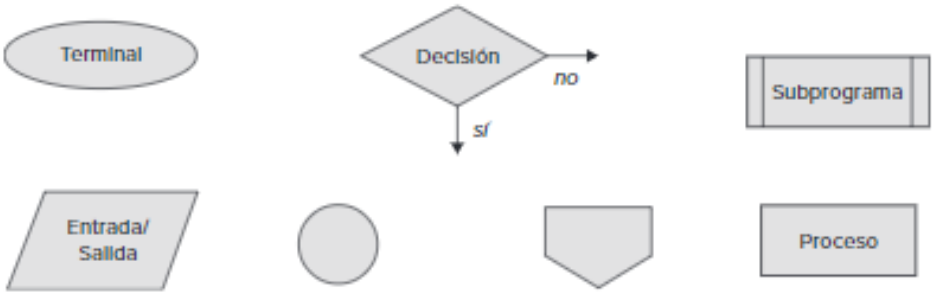


Figura 2.15. Plantilla típica para diagramas de flujo.

Se puede escribir más de un paso del algoritmo en una sola caja rectangular. El uso de flechas significa que la caja no necesita ser escrita debajo de su predecesora. Sin embargo, abusar demasiado de esta flexibilidad conduce a diagramas de flujo complicados e ininteligibles.

EJEMPLO 2.7

Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde una terminal. Un valor de cero —como entrada— indicará que se ha alcanzado el final de la serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, nuestro algoritmo en forma descriptiva sería:

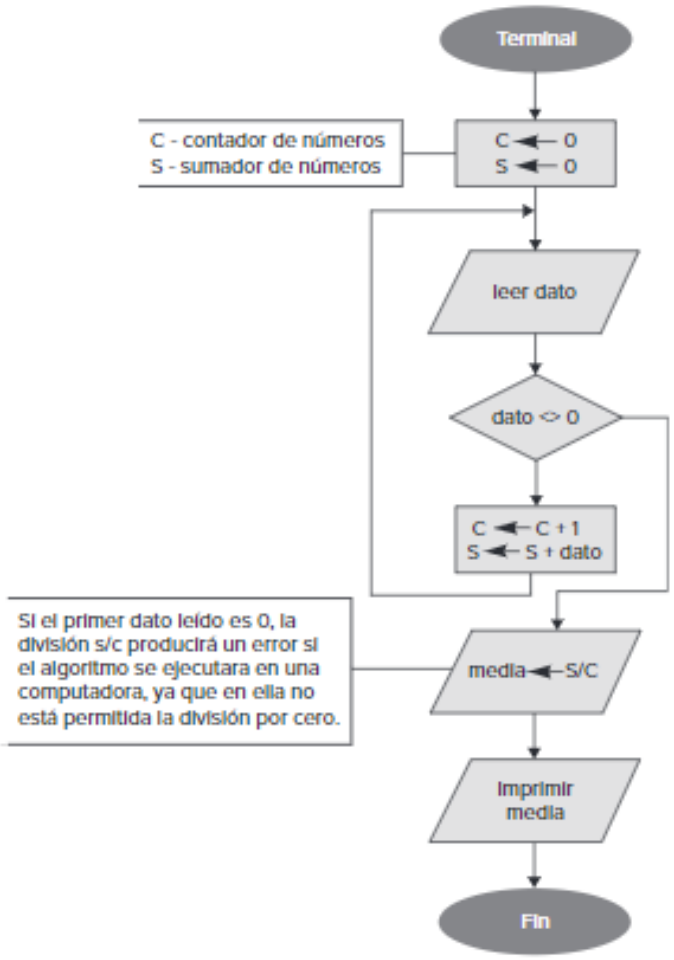
1. Inicializar contador de números C y variable suma S.
2. Leer un número.
3. Si el número leído es cero:
   - calcular la media;
   - imprimir la media;
   - fin del proceso.

   Si el número leído no es cero:
   - calcular la suma;
   - incrementar en uno el contador de números;
   - ir al paso 2.
4. Fin.

El refinamiento del algoritmo conduce a los pasos sucesivos necesarios para realizar las operaciones de lectura, verificación del último dato, suma y media de los datos.

Si el primer dato leído es 0, la división S/C produciría un error si el algoritmo se ejecutara en una computadora, ya que en ella no está permitida la división por cero.

Diagrama de flujo



Pseudocódigo

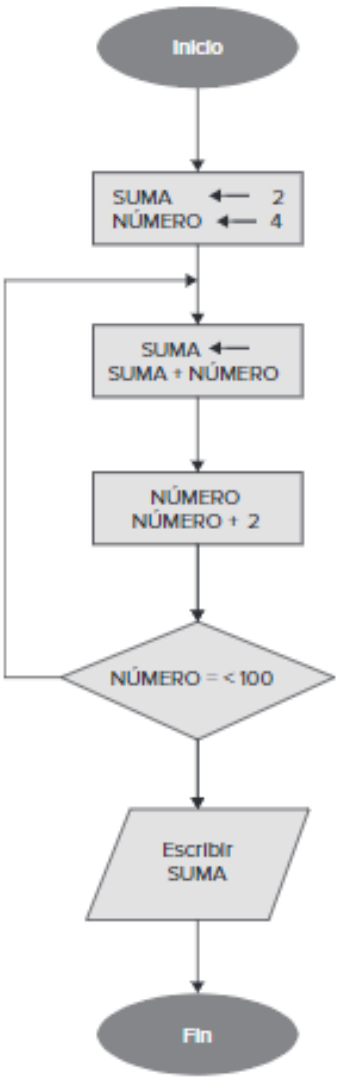
```
entero: dato, C
real: media, S

C ←0
S ←0
escribir('Datos numéricos;
         para finalizar se
         introduce 0')

repetir
   leer(dato)
   si dato <> = 0 entonces
      C ←C + 1
      S ←S + dato
   fin_si

hasta dato = 0
{Calcula la media y la escribe}
si (C > 0) entonces
   Media ←S/C
   escribir (Media)
fin_si
```

Cristian Palpanta 1095211-2202100206

EJEMPLO 2.8

Suma de los números pares comprendidos entre 2 y 100.

Diagrama de flujo



Pseudocódigo

```
entero:número,
Suma ← 2
número ← 4
mientras (número <= 100) hacer
   suma ← suma + número ←
   número + 2
fin mientras
escribe ('Suma pares entre 2 y 100 =', suma)
```

Cristian Pilapanta 1095211-2202100008

1095211-2202100008

EJEMPLO 2.9

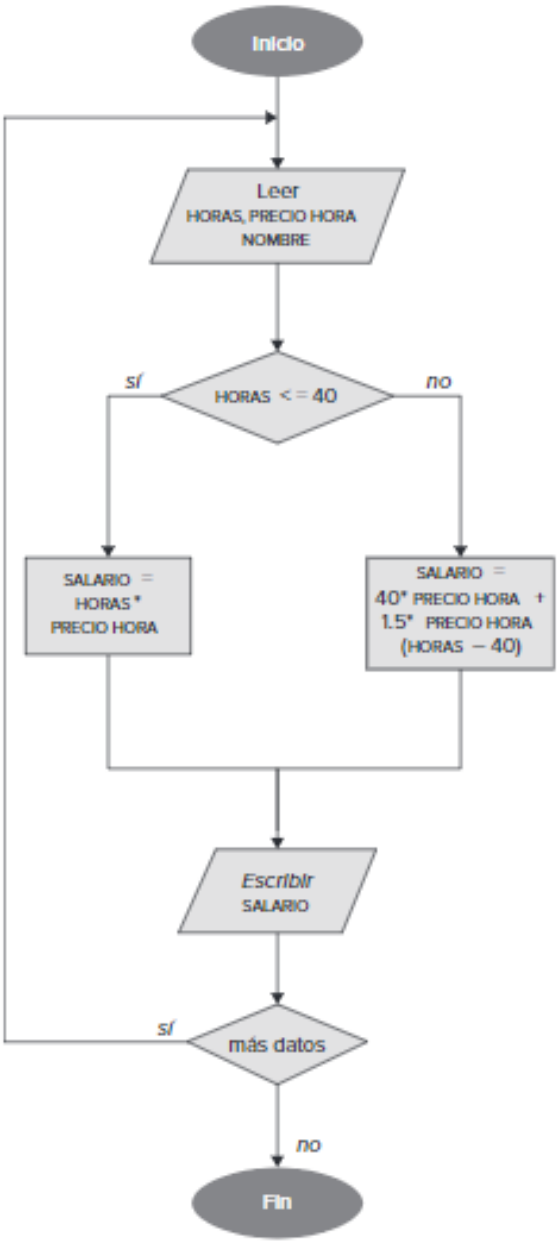
Se desea realizar el algoritmo que resuelva el siguiente problema: Cálculo de los salarios mensuales de los empleados de una empresa, sabiendo que estos se calculan con base en las horas semanales trabajadas y de acuerdo con un precio especificado por horas. Si se pasan de cuarenta horas semanales, las horas extraordinarias se pagarán a razón de 1.5 veces la hora ordinaria.

Los cálculos son:

1. Leer datos del archivo de la empresa, hasta que se encuentre la ficha final del archivo (HORAS, PRECIO_HORA, NOMBRE).
2. Si HORAS <= 40, entonces SALARIO es el producto de horas por PRECIO_HORA.
3. Si HORAS > 40, entonces SALARIO es la suma de 40 veces PRECIO_HORA más 1.5 veces PRECIO_HORA por (HORAS-40).

El diagrama de flujo completo del algoritmo y la codificación en pseudocódigo se indican a continuación:

Diagrama de flujo



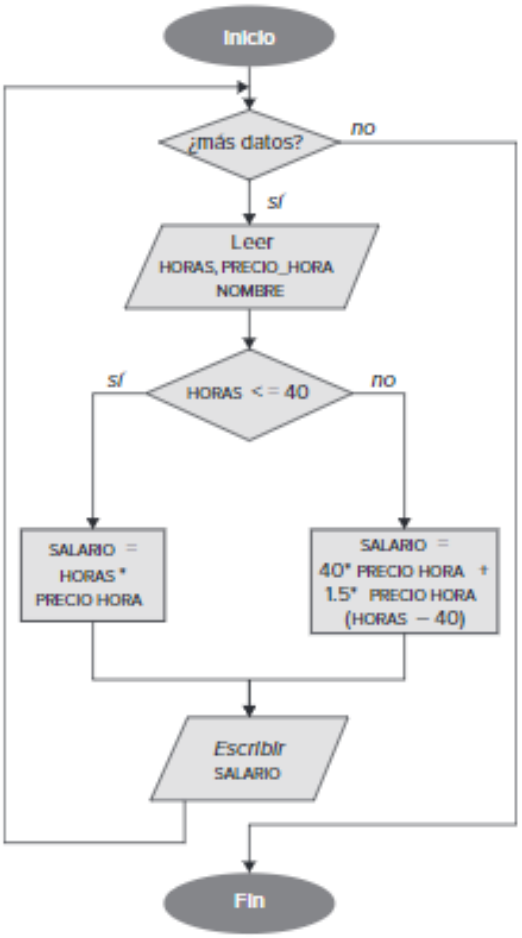
Pseudocódigo

```
real: horas, precioHora, salario
cadena: nombre
carácter: másDatos

inicio
escribir('Introducir horas, precio hora y
nombre')
repetir
   escribir ('Nombre')
   leer (Nombre)
   escribir ('Horas trabajadas')
   leer (horas)
   escribir ('Precio hora')
   leer (precio Hora)
   si (horas <= 40) entonces
      Salario ← horas * precio Hora
   sino
      Salario ← 40 * precio Hora +
      1.5 * (horas - 40) * precio hora
   fin si
   escribir ('Salario de', nombre, salario)
   escribir ('Más trabajadores S/N')
   leer (más Datos)
hasta más Datos = 'N'
fin
```

Cristian Palpanta 1006211-2202100308

Una variante también válida del diagrama de flujo anterior es:

Diagrama de flujo



Pseudocódigo

```
real: horas, precio Hora, salario
cadena: nombre
carácter: más Datos

inicio
más Datos ← 'S';
escribir ('Introducir horas, precio hora
          y nombre')

mientras (más Datos = 'S' o más Datos = 's')
          hacer
   escribir ('Nombre') leer (nombre)
   escribir ('Horas trabajadas') leer (horas)
   escribir ('Precio hora') leer (Precio Hora)
   si horas <= 40 entonces
      salario ← horas * precioHora
   sino
      salario ← 40 * precio Hora + 1.5 *
      (horas - 40) * precio Hora
   fin si
   escribir ('Salario de', nombre, salario)
   escribir ('Más trabajadores (S/N)')
   leer (más Datos)
fin mientras
fin
```

EJEMPLO 2.10

La escritura de algoritmos para realizar operaciones sencillas de conteo es una de las primeras cosas que una computadora puede aprender.

Supongamos que se proporciona una secuencia de números, tales como:

5 3 0 2 4 4 0 0 2 3 6 0 2

y desea contar e imprimir el número de ceros de la secuencia.

El algoritmo es muy sencillo, ya que sólo basta leer los números de izquierda a derecha mientras se cuentan los ceros. Utiliza como variable la palabra NÚMERO para los números que se examinan y TOTAL para el número de ceros encontrados. Los pasos a seguir son:

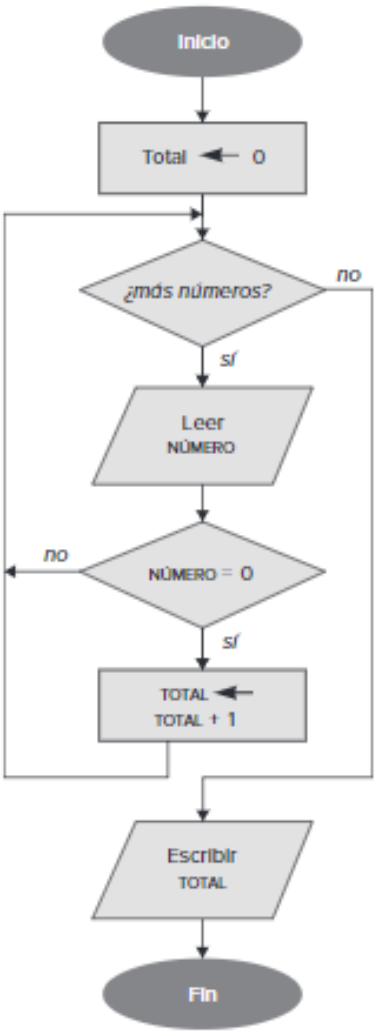
1. Establecer TOTAL a cero.
2. ¿Quedan más números a examinar?
3. Si no quedan números, imprimir el valor de TOTAL y fin.
4. Si existen más números, ejecutar los pasos 5 a 8.
5. Leer el siguiente número y dar su valor a la variable NÚMERO.

Cristian Palpanta 1065211-2202100000

6. Si NÚMERO = 0, incrementar TOTAL en 1.
7. Si NÚMERO <> 0, no modificar TOTAL.
8. Retornar al paso 2.

El diagrama de flujo y la codificación en pseudocódigo correspondiente es:

Diagrama de flujo



Pseudocódigo

```
entero: número, total
carácter: más Datos;

inicio
escribir ('Cuenta de ceros leídos del teclado')
más Datos ←'S';
total ←0

mientras (más Datos = 'S') o (más Datos = 's') hacer
   leer (número)
   si (número = 0)
      total ←total + 1
   fin si
   escribir ('Más números 'S/N")
   leer (más Datos)
fin mientras
escribir ('total de ceros =', total)
fin
```

EJEMPLO 2.11

Dados tres números, determinar si la suma de cualquier pareja de ellos es igual al tercer número. Si se cumple esta condición, escribir "Iguales" y, caso contrario, escribir "Distintas".

En el caso de que los números sean: 3 9 6

la respuesta es "Iguales", ya que 3 + 6 = 9. Sin embargo, si los números fueran:

2 3 4

el resultado sería "Distintas".

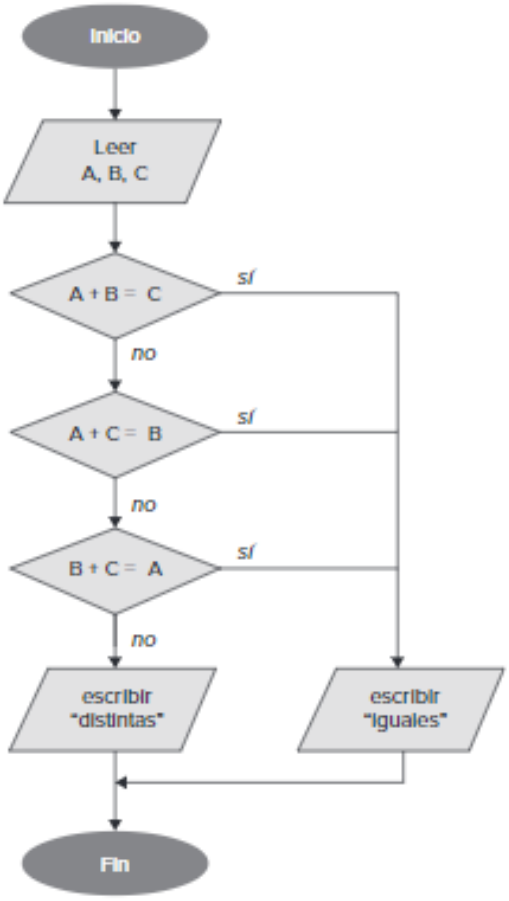
Cristian Pilapanta 1095211-2202100010

Para resolver este problema se puede comparar la suma de cada pareja con el tercer número. Con tres números solamente existen tres parejas distintas y el algoritmo de resolución del problema será fácil.

1. Leer los tres valores, A, B y C.
2. Si A + B = C escribir "Iguales" y parar.
3. Si A + C = B escribir "Iguales" y parar.
4. Si B + C = A escribir "Iguales" y parar.
5. Escribir "Distintas" y parar.

El diagrama de flujo y la codificación en pseudocódigo correspondiente es la figura 2.16.

Diagrama de flujo



Pseudocódigo

```
entero: a, b, c

inicio
escribir ('test con tres números:')
leer (a, b, c)

si (a + b = c) entonces
      escribir ('Son iguales', a,'+',b,'=',c)
   sino si (a + c = b) entonces
      escribir ('Son iguales', a,'+',c,'=',b)
   sino si (b + c = a) entonces
      escribir ('Son iguales', b,'+',c,'=',a)
   sino
      escribir ('Son distintas')
   fin si
   fin si
fin si
fin
```

Figura 2.16. Diagrama de flujo y codificación en pseudocódigo (Ejemplo 2.11).

2.7.3. Diagramas de Nassi-Schneiderman (N-S)

El diagrama N-S de Nassi Schneiderman —también conocido como diagrama de Chapin— es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas (figura 2.17). Las acciones sucesivas se escriben en cajas sucesivas y, como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja.

Un algoritmo se representa con un rectángulo en el que cada banda es una acción a realizar.

Cristian Pilapanta 1006211-2202100311

EJEMPLO 2.12

Escribir un algoritmo que lea el nombre de un empleado, las horas trabajadas, el precio por hora y calcule los impuestos a pagar (tasa = 25%) y el salario neto.

```
leer
nombre, horas, precio
```

```
calcular
salario ← horas * precio
```

```
calcular
impuestos ← 0.25 * salario
```

```
calcular
neto ← salario impuestos
```

```
escribir
nombre, salario, impuestos, neto
```

```
nombre del algoritmo
<acción 1>
<acción 2>
<acción 3>
...
fin
```

Cristian Pilapanta 1005211-2202100212

1005211220210012

Figura 2.17. Representación gráfica N-S de un algoritmo.

Otro ejemplo es la representación de la estructura condicional (figura 2.18).

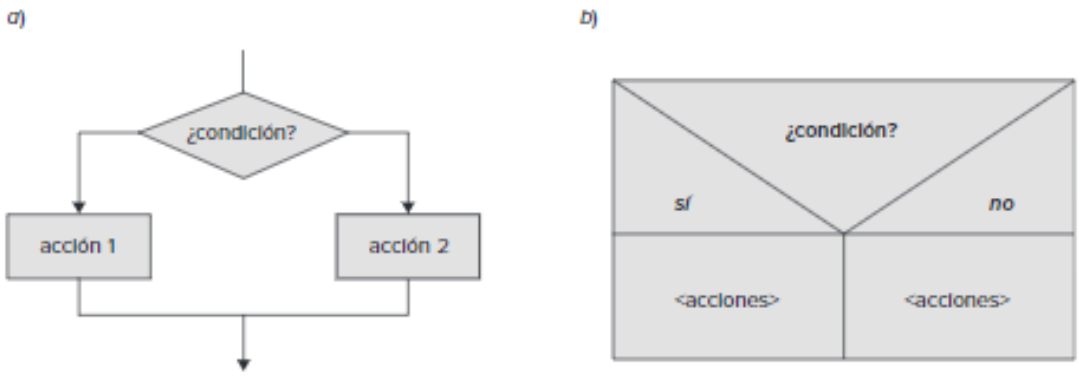


Figura 2.18. Estructura condicional o selectiva: *a*) diagrama de flujo, *b*) diagrama N-S.

EJEMPLO 2.13

Se desea calcular el salario neto semanal de un trabajador (en dólares o en euros) en función del número de horas trabajadas y la tasa de impuestos:

- *las primeras 35 horas se pagan a tarifa normal,*
- *las horas que pasen de 35 se pagan a 1.5 veces la tarifa normal,*
- *las tasas de impuestos son:*
  a) *los primeros 1 000 dólares son libres de impuestos,*
  b) *los siguientes 400 dólares tienen un 25 por 100 de impuestos,*
  c) *los restantes, 45% de impuestos,*
- *la tarifa horaria es de 15 dólares.*

También se desea escribir el nombre, salario bruto, tasas y salario neto (*este ejemplo se deja como ejercicio para el alumno*).

2.8. Herramientas y entornos de desarrollo de programación

La programación en C, C++, Java o cualquier otro lenguaje de programación requiere herramientas para la creación de un programa. Las herramientas más usuales son **editor, compilador** y **depurador de errores** y puesta a punto del programa, aunque existen otras herramientas, sobre todo en el caso de desarrollo profesional. Estas herramientas pueden ser independientes y utilizadas de esta forma o bien estar incluidas en entornos de desarrollo integrados y utilizadas como un todo. En el aprendizaje profesional se recomienda conocer ambas categorías de herramientas y, a medida que las vaya dominando, seleccionar cuáles considera las más idóneas para su trayectoria profesional.

Cristian Pilapanta 1086211-2200100013

2.8.1. Editores de texto

Un editor de textos es un programa de aplicación que permite escribir programas. Los editores que sirven para la escritura de programas en lenguajes de alto nivel son diferentes de los procesadores de texto tradicionales como Word de Microsoft, Google Docs o Zoho.

Un editor de software, ya sea independiente o integrado, es un entorno de desarrollo que normalmente debe proporcionar las características adecuadas para la adaptación de las normas de escritura de la sintaxis del lenguaje de programación correspondiente y también algunas propiedades relacionadas con la sintaxis de este lenguaje, que reconozca sangrados de línea y que reconozca y complete automáticamente palabras clave (reservadas) del lenguaje después de que los programadores hayan tecleado los primeros caracteres de la palabra.

Un editor de texto clásico es NotePad, que permite crear (escribir) un programa en Java o C++ siguiendo las reglas o sintaxis del lenguaje; otro editor típico es Edit (Edit.com) del antiguo sistema operativo MSDOS. El editor debe escribir el programa fuente siguiendo las reglas de sintaxis del lenguaje de programación y luego guardarlo en una unidad de almacenamiento como archivo de texto. Así, por ejemplo, en el caso de Java, un programa que se desea llamar `MiPrimerPrograma` (el nombre de una clase en Java) se debe guardar después de escribir en un archivo de texto denominado `MiPrimerPrograma.java` (nombre de la clase, una característica específica de Java). Antiguamente se utilizaban editores como Emacs, JEdit o TextPad.

2.8.2. Programa ejecutable

El programa o archivo ejecutable es el archivo binario (código máquina) cuyo contenido es interpretado por la computadora como un programa. El ejecutable contiene instrucciones en código máquina de un procesador específico, en los casos de lenguajes como C o C++, ya que se requiere un compilador diferente para cada tipo de CPU, o bien *bytecode* en Java, que es el código máquina que se produce cuando se compila un programa fuente Java y que es el lenguaje máquina de la máquina virtual (JVM), que es independiente de cualquier tipo de CPU. En el ejemplo anterior y en el caso de Java, el código traducido por el compilador viene en *bytecode* y se almacena en el compilador con el nombre `MiPrimerPrograma.class`.

El intérprete Java traduce cada instrucción en *bytecode* en un tipo específico de lenguaje máquina de la CPU y a continuación ejecuta la instrucción (desde un punto de vista práctico el compilador una vez que ha obtenido el código máquina *bytecode*, utiliza un cargador, que es un programa que recibe a su vez las funciones o clases correspondientes de una biblioteca Java y la salida alimenta al intérprete que va ejecutando las sucesivas instrucciones). En el caso de Java se requiere un tipo diferente de intérprete para cada procesador o CPU específica. Sin embargo, los intérpretes son programas más sencillos que los compiladores, pero como el intérprete Java traduce las instrucciones en *bytecodes* una detrás de otras, el programa Java se ejecuta más lentamente.

Los programas ejecutables pueden ser portables (se pueden ejecutar en diferentes plataformas) o no portables (están destinados a una plataforma concreta).

2.8.3. Proceso de compilación/ejecución de un programa

Las computadoras sólo entienden el lenguaje máquina. Por consiguiente, para ejecutar un programa con éxito, el código fuente (el programa escrito en un lenguaje de programación C/C++, Java o C#) o programa fuente, se debe traducir al lenguaje máquina o de la máquina mediante un compilador o, en su caso, un intérprete. El proceso de un programa escrito en un lenguaje de programación de alto nivel consta de cinco etapas: editar, compilar, enlazar, cargar y ejecutar, aunque según el tipo de lenguaje de programación C/C++ o Java, alguna de las etapas puede descomponerse en otras (figura 2.19).

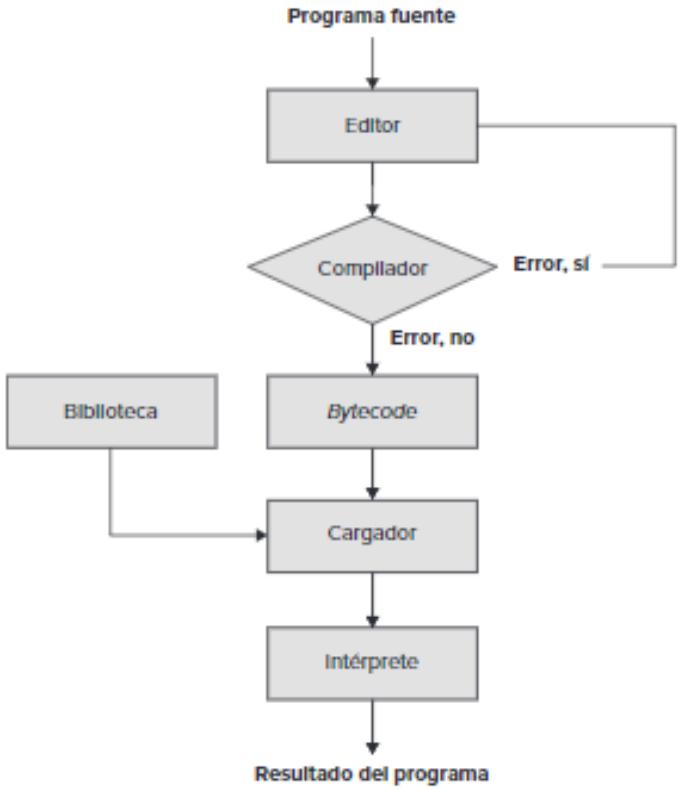


Figura 2.19. Proceso de ejecución de un programa en Java.

Existen dos métodos para procesar programas completos (compilación y ejecución). Uno son los programas de consola, normalmente conocidos como consola de línea de comandos, que son herramientas en las que los programadores deben teclear los comandos (las órdenes) en una consola o ventana Shell y ejecutar paso a paso las diferentes etapas de compilación. El segundo método es el más utilizado ya que suele ser más cómodo de usar: son los EDI

Cristian Pilapanta 1095211-2202100014

(*integrated development environment*), entorno de desarrollo integrado, que permiten la edición, compilación y ejecución de un programa de modo directo.

Aunque los EDI son más fáciles de aprender, puede resultar más tedioso su uso para el caso de programas pequeños, por lo que le aconsejamos que aprenda a manejar ambos métodos. Aunque lo más fácil sea casi siempre el EDI, habrá ocasiones en que puede resultarle mejor el uso de la consola de línea de comandos para aprendizaje o incluso para desarrollos muy profesionales.

Consola de línea de comandos

La compilación con la consola de línea de comandos es el procedimiento más antiguo y clásico de la compilación/ejecución de un programa fuente. Con este método se edita el programa fuente con un editor de archivos de texto como, por ejemplo, el bloc de notas del sistema operativo Windows (Notepad.exe) o el Edit (Edit.com) del sistema operativo MSDOS; una vez editado el programa fuente con el editor se compila el programa con el compilador. El editor y el compilador se ejecutan directamente desde la línea de comandos del sistema operativo en la ventana de comandos del sistema operativo.

En la fase de aprendizaje de programación se suele recomendar, en la etapa inicial, el uso de la edición, compilación y ejecución desde la ventana o consola de la línea de comandos, de modo que una vez adquirida la destreza y aprendido el mecanismo completo del proceso de ejecución, entonces se puede pasar a trabajar con un entorno de desarrollo integrado profesional.

En los últimos años han aparecido entornos de desarrollo profesionales excelentes y en muchos casos gratuitos, como es el caso de Eclipse y de NetBeans, con la gran ventaja de que en muchas ocasiones sirven para diferentes lenguajes de programación.

Cristian Pilapanta 1096211-2202100015

2.8.4. Entorno de desarrollo Integrado

El **entorno de desarrollo integrado** (**EDI**) contiene herramientas que soportan el proceso de desarrollo de software. Se compone de un **editor** para escribir y editar programas, un **compilador**, un **depurador** para detectar errores lógicos (errores que originan una ejecución no correcta del programa) y un constructor de interfaz gráfico de usuario (GUI). Además, suelen incluir herramientas para compilar e interpretar, en su caso, los programas fuente. Los EDI pueden estar orientados a uno o varios lenguajes de programación, aunque generalmente están orientados a uno solo. Existen entornos de desarrollo para casi todos los lenguajes de programación, como C, C++, Python, Java, C#, Delphi, Visual Basic, Pascal, ObjectiveC (el lenguaje de desarrollo de aplicaciones de Apple para teléfonos inteligentes iPhone), etcétera.

Algunos entornos de desarrollo integrados populares son: 1) En C++: Microsoft Visual Studio 2010, DevC++, NetBeans, Eclipse; 2) En Java: Eclipse, NetBeans, JBuilder, JCreator, JGrasp, BlueJ y Java Development Kit (JDK). Cualquiera de estos entornos de desarrollo le será a usted de gran utilidad, por lo que le aconsejamos que descargue e instale el entorno elegido o aquel recomendado por su profesor o maestro en clase.

2.8.5. Panorama de los entornos de programación

Las herramientas de programación que soportan los entornos integrados de desarrollo (EID, IDE en inglés) son en la actualidad muy numerosos:

- Editores de texto (para escritura de programas).
- Compiladores e intérpretes (traducción de los programas fuente a código máquina).
- Depuradores (detección de errores) y analizadores estáticos para detectar patrones típicos en códigos erróneos, éstos advierten de potenciales problemas.
- Analizadores en tiempo de ejecución (permiten estudiar la eficiencia de los programas).
- Analizadores de cobertura (para aseguramiento de que el código ha sido puesto a prueba).
- Sistemas de control de versiones (permiten la colaboración de programadores en un mismo programa, al facilitar la recuperación de cualquier versión de programa, creación de desarrollo de programas en paralelo).
- Generadores de documentación (generan documentos HTML a partir de comentarios en los programas).
- Otros.

Tabla 2.3. Entornos de desarrollo integrados populares.

BlueJ	(www.blueJ.org)
NetBeans	(www.netbeans.org)
JBuilder	(www.borland.com)
Eclipse	(www.eclipse.org)
JCreator	(www.jcreator.com)
JEdit	(www.jedit.org)
JGrasp	(www.jgrasp.org)
Dev-C++	(http://www.bloodshed.net/devcpp.html)
Microsoft Visual C++	(http://www.microsoft.com)

Actividades de aprendizaje

2.1. Reconozca los conceptos básicos de programa, programación, paradigmas de programación utilizando herramientas teóricas y prácticas.

2.2. Conozca el entorno de un lenguaje de programación: entorno de desarrollo integrado y consola de línea de comandos para la compilación y ejecución de programas.

2.3. Busque y analice información necesaria para instalar y configurar el compilador del lenguaje de programación a utilizar (lenguajes C, C++ y Java).

2.4. Practique con editores, compiladores e intérpretes.

2.5. Practique con la consola de línea de comandos y entornos de desarrollo integrado de C, C++ y Java.

2.6. Explique los conceptos básicos para la formulación de algoritmos, así como sus ventajas y desventajas.

2.7. Genere un catálogo de problemas para su análisis y solución.

2.8. Resuelva y analice problemas cotidianos.

2.9. Investigue los diferentes métodos para representar un algoritmo: diagrama de flujo, N-S (Nassi-Schneiderman).

Actividades complementarias

2.1. Imagine una aplicación simple de ingeniería (por ejemplo, resolver un sistema de ecuaciones, cálculo de la superficie de una figura geométrica).

2.2. Proponga las etapas para la resolución del problema.

2.3. Considere la descomposición del problema en módulos.

2.4. Escriba el algoritmo que resuelve el problema en un diagrama de flujo y en un pseudocódigo.

2.5. Busque en Internet el concepto de programación estructurada y amplíe los conceptos explicados en clase y en el libro.

2.6. Busque en Internet información sobre programación orientada a objetos y lenguajes de programación orientada a objetos.

2.7. Busque y estudie la información encontrada sobre programación modular y funciones.

2.8. Escriba un algoritmo para hacer guacamole; primero en texto narrado y después en pseudocódigo y diagrama de flujo.

2.9. Escriba un algoritmo para fabricar tequila reposado, primero en texto narrado y después en pseudocódigo y diagrama de flujo.

Cristian P Espana 1096211-2202100016

2.10. Escriba un algoritmo para ir al cine o a un concierto de Maná en Veracruz el próximo domingo comprando la entrada en un comercio electrónico de Internet (se supone que puede estar en Veracruz o en otra ciudad de México).

2.11. Escriba un algoritmo para efectuar una compra de un libro de programación por Internet en un comercio electrónico; por ejemplo en la tienda Amazon.

2.12. Busque información sobre diagramas N-S con objeto de ampliar los conceptos explicados en el libro y diseñar los algoritmos de "guacamole" y "tequila reposado" y "compra de un libro en Amazon" mediante diagramas N-S.

2.13. Busque y descargue alguna herramienta de software libre (preferentemente gratuita) para realización de diagramas de flujo. Una vez que la tenga instalada trate de hacer los gráficos de los algoritmos 5, 6 y 7, comparando con las realizadas por usted mismo manualmente.

2.14. Trate de buscar herramientas para la realización de diagramas N-S.

Resumen

Un método general para la resolución de un problema con computadora tiene las siguientes fases:

1. Análisis del programa.
2. Diseño del algoritmo.
3. Codificación.
4. Compilación y ejecución.
5. Verificación.
6. Documentación y mantenimiento.

El sistema más idóneo para resolver un problema es descomponerlo en módulos más sencillos y luego, mediante diseños descendentes y refinamiento sucesivo, llegar a módulos fácilmente codificables. Estos módulos se deben codificar con las estructuras de control de programación estructurada.

1. Secuenciales: las instrucciones se ejecutan sucesivamente una después de otra.
2. Repetitivas: una serie de instrucciones se repiten una y otra vez hasta que se cumple una cierta condición.
3. Selectivas: permite elegir entre dos alternativas (dos conjuntos de instrucciones), dependiendo de una condición determinada.

Ejercicios

2.1. Diseñar una solución para resolver cada uno de los siguientes problemas y tratar de refinar sus soluciones mediante algoritmos adecuados:

a) Realizar una llamada telefónica desde un teléfono público.
b) Cocinar una tortilla.
c) Arreglar un pinchazo de una bicicleta.
d) Freír un huevo.

2.2. Escribir un algoritmo para:

a) Sumar dos números enteros.
b) Restar dos números enteros.
c) Multiplicar dos números enteros.
d) Dividir un número entero por otro.

2.3. Escribir un algoritmo para determinar el máximo común divisor de dos números enteros (MCD) por el algoritmo de Euclides:

- Dividir el mayor de los dos enteros positivos por el más pequeño.
- Luego dividir el divisor por el resto.
- Continuar el proceso de dividir el último divisor por el último resto hasta que la división sea exacta.
- El último divisor es el mcd.

2.4. Diseñar un algoritmo que lea y visualice una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe visualizar. Visualizar el número de valores leídos.

Cristian Palapanta 1086211-2200100016

2.5. Diseñar un algoritmo que visualice y sume la serie de números 3, 6, 9, 12..., 99.

2.6. Escribir un algoritmo que lea cuatro números y a continuación visualice el mayor de los cuatro.

2.7. Diseñar un algoritmo que lea tres números y descubra si uno de ellos es la suma de los otros dos.

2.8. Diseñar un algoritmo para calcular la velocidad (en m/s) de los corredores de la carrera de 1500 metros. La entrada consistirá en parejas de números (minutos, segundos) que dan el tiempo del corredor; por cada corredor, el algoritmo debe visualizar el tiempo en minutos y segundos, así como la velocidad media.

Ejemplo de entrada de datos: (3,53) (3,40) (3,46) (3,52) (4,0) (0,0); el último par de datos se utilizará como fin de entrada de datos.

2.9. Diseñar un algoritmo para determinar los números primos iguales o menores que *N* (leído del teclado). (Un número primo sólo puede ser divisible por él mismo y por la unidad.)

2.10. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura ($S = 1/2$ base $\times$ altura).

2.11. Calcular y visualizar la longitud de la circunferencia y el área de un círculo de radio dado.

2.12. Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.

2.13. Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* (capicúa) es una palabra que se lee igual en ambos sentidos como "radar".

2.14. Escribir un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. Por ejemplo, `"Mortimer"` contiene dos `"m"`, una `"o"`, dos `"r"`, una `"i"`, una `"t"` y una `"e"`.

2.15. Muchos bancos y cajas de ahorro calculan los intereses de las cantidades depositadas por los clientes diariamente según las premisas siguientes. Un capital de 1000 euros, con una tasa de interés de 6%, renta un interés en un día de 0.06 multiplicado por 1000 y dividido por 365. Esta operación producirá 0.16 euros de interés y el capital acumulado será 1000,16. El interés para el segundo día se calculará multiplicando 0.06 por 1000 y dividiendo el resultado por 365. Diseñar un algoritmo que reciba tres entradas: el capital a depositar, la tasa de interés y la duración del depósito en semanas, y calcular el capital total acumulado al final del periodo especificado.

Cristian Pilapanta 1096211-2202100018

1095211220210019

CAPÍTULO 3

Estructura general de un programa

3.1. Concepto de programa
3.2. Estructura de un programa
3.3. Instrucciones y tipos de instrucciones
3.4. Elementos básicos de un programa
3.5. Datos, tipos de datos y operaciones primitivas
3.6. Constantes y variables
3.7. Expresiones y operadores
3.8. Funciones internas
3.9. Operación de asignación
3.10. Operadores avanzados
3.11. Entrada y salida de información
3.12. Escritura de algoritmos/programas
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Cristian Pilapanta 1095211-2202100320

1095211220210032

INTRODUCCIÓN

En los capítulos anteriores se ha visto la forma de diseñar algoritmos para resolver problemas con computadora. En este capítulo se introduce al proceso de la programación que se manifiesta esencialmente en los programas.

El *concepto de programa* como un conjunto de instrucciones y sus tipos constituye la parte fundamental del capítulo. La *descripción de los elementos básicos* de programación, que se encontrarán en casi todos los programas: interruptores, contadores, totalizadores, etcétera, junto con las normas elementales para la escritura de algoritmos y programas, conforman el resto del capítulo.

En este capítulo se examinan los importantes conceptos de datos, constantes y variables, expresiones, operaciones de asignación y la manipulación de las entradas y salidas de información, así como la realización de las funciones internas como elemento clave en el manejo de datos. Por último, se describen reglas de escritura y de estilo para la realización de algoritmos y su posterior conversión en programas.

3.1. Concepto de programa

Un *programa de computadora* es un conjunto de instrucciones –órdenes dadas a la máquina– que producirán la ejecución de una determinada tarea. En esencia, *un programa es un medio para conseguir un fin*. El fin será probablemente definido como la información necesaria para solucionar un problema.

El *proceso de programación*, figura 3.1, es, por consiguiente, un proceso de solución de problemas –como ya se vio en el capítulo 2– y el desarrollo de un programa requiere las siguientes fases:

1. *Definición y análisis del problema.*
2. *Diseño de algoritmos:*
   - Diagrama de flujo.
   - Diagrama N-S.
   - Pseudocódigo.
3. *Codificación del programa.*
4. *Depuración y verificación del programa.*
5. *Documentación.*
6. *Mantenimiento.*

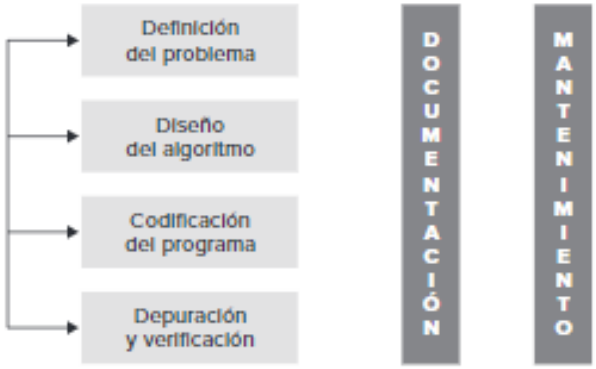


Figura 3.1. El proceso de la programación.

Las fases 1 y 2 han sido analizadas en los capítulos anteriores y son el objetivo fundamental de este texto; sin embargo, dedicaremos atención a lo largo del libro (capítulo 13) y en los apéndices, a las fases 3, 4, 5 y 6 aunque éstas son propias de libros específicos sobre lenguajes de programación.

3.2. Estructura de un programa

Tras la decisión de desarrollar un programa, el programador necesita establecer el conjunto de especificaciones que éste debe contener: *entrada*, *salida* y *algoritmos de resolución*, que incluirán las técnicas para obtener las salidas a partir de las entradas.

Conceptualmente un programa puede ser considerado una caja negra, como se muestra en la figura 3.2. La caja negra o el algoritmo de resolución en realidad son el conjunto de códigos que transforman las entradas del programa (*datos*) en salidas (*resultados*). El programador debe establecer de dónde provienen las entradas al programa; que en cualquier caso, procederán de un dispositivo de entrada –teclado, disco, etcétera–. El proceso de introducir la información –datos– en la memoria de la computadora se denomina *entrada de datos*, operación de *lectura* o acción de `leer`.

Las salidas de datos se deben presentar en dispositivos periféricos de salida: *pantalla*, *impresoras*, *discos*, etcétera. La operación de *salida de datos* se conoce también como *escritura* o acción de escribir.

Cristian Pilapanta 1095211-2202100924

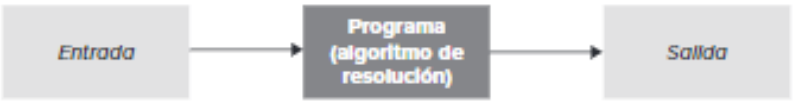


Figura 3.2. Bloques de un programa.

3.3. Instrucciones y tipos de instrucciones

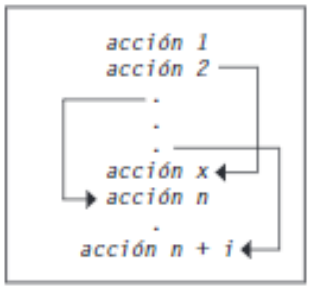
El proceso de diseño del algoritmo o posterior codificación del programa consiste en definir las acciones o instrucciones que resolverán el problema. Las *acciones* o *instrucciones* se deben escribir y luego almacenar en memoria en el mismo orden en el que han de ejecutarse; es decir, *en secuencia*. Un programa puede ser lineal o no lineal. Es *lineal* si las instrucciones se ejecutan secuencialmente, sin bifurcaciones, decisión ni comparaciones.

```
instrucción 1
instrucción 2
      .
      .
      .
instrucción n
```

En el caso del algoritmo las instrucciones se suelen conocer como *acciones*, y se tendría:

```
acción 1
acción 2
   .
   .
   .
acción n
```

Un programa es *no lineal* cuando se interrumpe la secuencia mediante instrucciones de bifurcación.



3.3.1. Tipos de Instrucciones

Las instrucciones disponibles en un lenguaje de programación dependen del tipo de lenguaje. Por ello, en este apartado estudiaremos las instrucciones —acciones— básicas que se pueden implementar de modo general en un

Cristian Palpanta 1095211-2202100027

algoritmo y que esencialmente soportan todos los lenguajes. Dicho de otro modo, las instrucciones básicas son independientes del lenguaje. La clasificación más usual, desde el punto de vista anterior, es:

1. *Instrucciones de inicio/fin.*
2. *Instrucciones de asignación.*
3. *Instrucciones de lectura.*
4. *Instrucciones de escritura.*
5. *Instrucciones de bifurcación.*

Algunas de estas instrucciones se recogen en la tabla 3.1.

Tabla 3.1. Instrucciones/Acciones básicas.

Tipo de instrucción	Pseudocódigo inglés	Pseudocódigo español
comienzo de proceso	`begin`	`inicio`
fin de proceso	`end`	`fin`
entrada (lectura)	`read`	`leer`
salida (escritura)	`write`	`escribir`
asignación	`A ← 5`	`B ← 7`

Cristian P Alapanta 1095211-2202100028

3.3.2. Instrucciones de asignación

Para una mejor comprensión por parte del lector, repasaremos su funcionamiento con ejemplos:

a) `A ← 80` la variable `A` toma el valor de 80.

b) ¿Cuál será el valor que tomará la variable `C` tras la ejecución de las siguientes instrucciones?

```
A ← 12
B ← A
C ← B
```

`A` contiene 12, `B` contiene 12 y `C` contiene 12.

1095211-2202100028

Nota

Antes de la ejecución de las tres instrucciones, el valor de `A`, `B` y `C` es indeterminado. Si desea darles un valor inicial, habrá que hacerlo explícitamente, incluso cuando este valor sea 0. Es decir, será necesario definir e inicializar las instrucciones.

```
A ← 0
B ← 0
C ← 0
```

c) ¿Cuál es el valor de la variable `AUX` al ejecutarse la instrucción 5?

```
1. A   ← 10
2. B   ← 20
3. AUX ← A
4. A   ← B
5. B   ← AUX
```

- En la instrucción 1, `A` toma el valor `10`.
- En la instrucción 2, `B` toma el valor `20`.
- En la instrucción 3, `AUX` toma el valor anterior de `A`, o sea `10`.

- En la instrucción 4, A toma el valor anterior de B, o sea 20.
- En la instrucción 5, B toma el valor anterior de AUX, o sea 10.
- Tras la instrucción 5, AUX sigue valiendo 10.

d) ¿Cuál es el significado de N ← N + 5 si N tiene el valor actual de 2?

```
N ←N + 5
```

Se realiza el cálculo de la expresión N + 5 y su resultado 2 + 5 = 7 se asigna a la variable situada a la izquierda, es decir, N tomará un nuevo valor 7.

Se debe pensar en la variable como en una posición de memoria, cuyo contenido puede variar mediante instrucciones de asignación (un símil suele ser un buzón de correos, donde el número de cartas depositadas en él variará según el movimiento diario del cartero de introducción de cartas o del dueño del buzón de extracción de dichas cartas).

3.3.3. Instrucciones de lectura de datos (entrada)

Esta instrucción lee datos de un dispositivo de entrada. ¿Cuál será el significado de las instrucciones siguientes?

a) `leer (NÚMERO, HORAS, TASA)`

Leer del terminal los valores NÚMERO, HORAS y TASAS, archivándolos en la memoria; si los tres números se teclean en respuesta a la instrucción son 12325, 32, 1200, significaría que se han asignado a las variables esos valores y equivaldría a la ejecución de las instrucciones.

```
NÚMERO ←12325
HORAS  ←32
TASA   ←1200
```

b) `leer (A, B, C)`

Si se leen del terminal 100, 200, 300, se asignarían a las variables los siguientes valores:

```
A = 100
B = 200
C = 300
```

3.3.4. Instrucciones de escritura de resultados (salida)

Estas instrucciones se escriben en un dispositivo de salida. Explicar el resultado de la ejecución de las siguientes instrucciones:

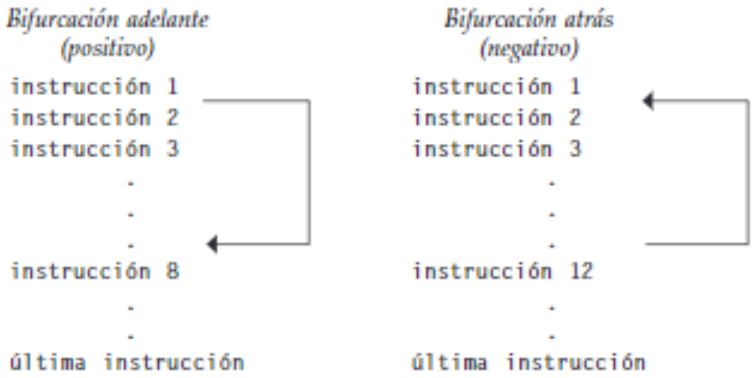
```
A ←100
B ←200
C ←300
escribir (A, B, C)
```

Se visualizarían en la pantalla o imprimirían en la impresora los valores 100, 200 y 300 que contienen las variables A, B y C.

3.3.5. Instrucciones de bifurcación

El desarrollo lineal de un programa se interrumpe cuando se ejecuta una bifurcación. Las bifurcaciones pueden ser, según el punto del programa a donde se bifurca, hacia *adelante* o hacia *atrás*.

Cristian Pilapanta 1006211-2202100029



Las bifurcaciones en el flujo de un programa se realizarán de modo condicional en función del resultado de la evaluación de la condición.

Bifurcación incondicional: la bifurcación se realiza siempre que el flujo del programa pase por la instrucción sin necesidad del cumplimiento de ninguna condición (figura 3.3).

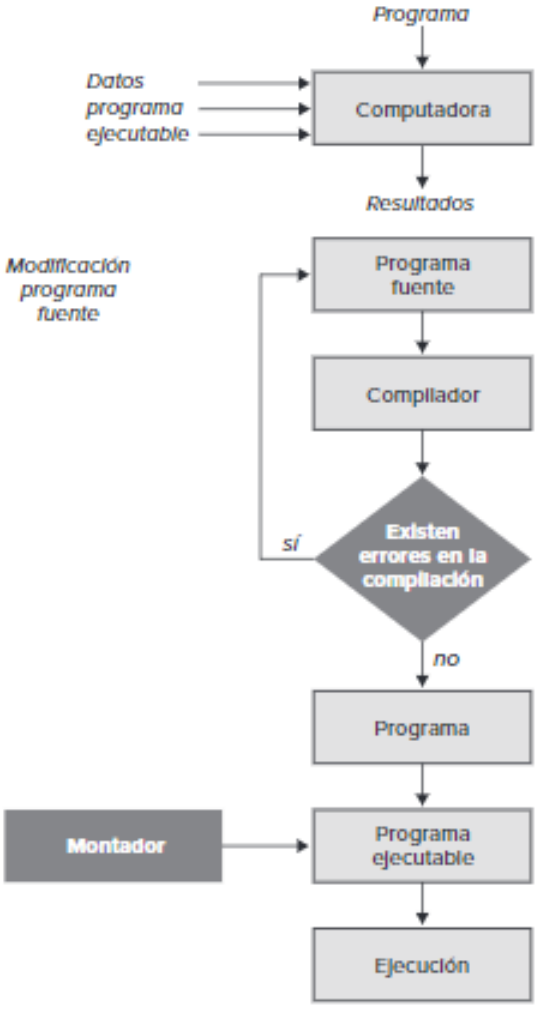


Figura 3.3. Fases de la ejecución de un programa.

Bifurcación condicional: la bifurcación depende del cumplimiento de una determinada condición. Si se cumple la condición, el flujo sigue ejecutando la acción F2; si no se cumple, se ejecuta la acción F1 (figura 3.4).

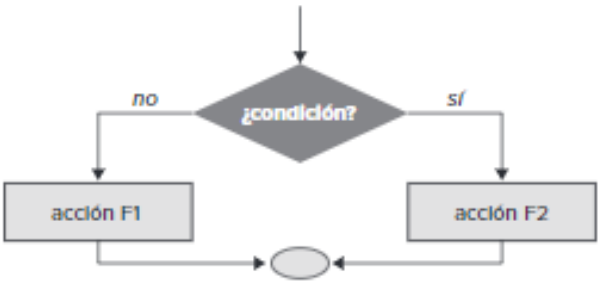


Figura 3.4. Bifurcación condicional.

3.4. Elementos básicos de un programa

En programación se debe marcar la diferencia entre el diseño del algoritmo y su implementación en un lenguaje específico. Por ello, es necesario distinguir claramente entre los conceptos de programación y el medio en que ellos se implementan en un lenguaje específico. Sin embargo, una vez que se comprenda cómo utilizar los conceptos de programación, la enseñanza de un nuevo lenguaje es relativamente fácil.

Los lenguajes de programación —como los restantes lenguajes— tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se combinen. Estas reglas se denominan *sintaxis* del lenguaje; solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis serán rechazados por la máquina. Los elementos básicos constitutivos de un programa o algoritmo son:

- *Palabras reservadas* (`inicio`, `fin`, `si-entonces...`, etcétera).
- *Identificadores* (nombres de variables esencialmente, procedimientos, funciones, nombre del programa, etcétera).
- *Caracteres especiales* (coma, apóstrofo, etcétera).
- *Constantes.*
- *Variables.*
- *Expresiones.*
- *Instrucciones.*

Además de estos elementos básicos existen otros que forman parte de los programas, cuya comprensión y funcionamiento será vital para el correcto diseño de un algoritmo y naturalmente la codificación del programa. Estos elementos son:

- *Bucles* (*Lazos*)
- *Contadores*
- *Acumuladores*
- *Interruptores*
- *Estructuras:*
  1. Secuenciales.
  2. Selectivas.
  3. Repetitivas.

El amplio conocimiento de todos los elementos de programación y el modo de su integración en los programas constituyen las técnicas que todo buen programador debe conocer.

Cristian Palapanta 1095211-2202100331

3.5. Datos, tipos de datos y operaciones primitivas

El primer objetivo de toda computadora es el manejo de la información o datos. Éstos pueden ser las cifras de ventas de un supermercado o las calificaciones de una clase. Un *dato* es la expresión general que describe los objetos con los cuales opera una computadora; la mayoría de las computadoras pueden trabajar con varios tipos (modos) de datos. Los algoritmos y los programas correspondientes operan sobre esos tipos de datos.

La acción de las instrucciones ejecutables de las computadoras se refleja en cambios en los valores de las partidas de datos. Los datos de entrada se transforman por el programa, después de las etapas intermedias, en datos de salida.

En el proceso de resolución de problemas el diseño de la estructura de datos es tan importante como el diseño del algoritmo y del programa que se basa en el mismo.

Un programa de computadora opera sobre datos (almacenados internamente en la memoria almacenados en medios externos como discos, memorias USB, memorias de teléfonos celulares, en la nube, etcétera, o bien introducidos desde un dispositivo como un teclado, un escáner o un sensor eléctrico). En los lenguajes de programación los datos deben ser de un *tipo específico;* el tipo de datos determina cómo se representan esos datos en la computadora y los diferentes procesos que dicha computadora realiza con ellos.

Tipo de datos

Conjunto específico de valores de los datos y un conjunto de operaciones que actúan sobre ellos.

Cristian Pilapana 1096211-2202100232

Existen dos tipos de datos: *básicos, incorporados o integrados* (estándar) que se incluyen en los lenguajes de programación y *definidos por el programador o por el usuario*.

Además de los datos básicos o simples se pueden construir otros datos a partir de éstos, y se obtienen los datos compuestos, o datos agregados, tales como **estructuras, uniones, enumeraciones** (*subrango*, como caso particular de las enumeraciones, al igual de lo que sucede en Pascal), **vectores** o **matrices/tablas** y **cadenas**, ***arrays*** **o** ***arreglos;*** también existen otros datos especiales en lenguajes como C y C++, denominados **punteros** (**apuntadores**) y **referencias**.

Los distintos tipos de datos se representan en diferentes formas en la computadora. A nivel de máquina, un dato es un conjunto o secuencia de bits (dígitos 0 o 1). Los lenguajes de alto nivel permiten basarse en abstracciones e ignorar los detalles de la representación interna. Los tipos de datos básicos son:

- **Numéricos** (*entero, real*).
- **Lógicos** (*booleano*).
- **Carácter** (*carácter, cadena*).

Existen algunos lenguajes de programación —Fortran esencialmente— que admiten otros tipos de datos: **complejos**, que permiten tratar los números complejos; y otros —como Pascal— que también permiten declarar y definir sus propios tipos de datos: **enumerados** (*enumerated*) y **subrango** (*subrange*).

3.5.1. Datos numéricos

El tipo *numérico* es el conjunto de valores numéricos; éstos pueden representarse en dos formas:

- Tipo numérico *entero* (*integer*).
- Tipo numérico *real* (*real*).

Enteros: este tipo de datos es un subconjunto finito de los números enteros; son números completos, no tienen componentes fraccionarios o decimales y pueden ser negativos o positivos. Ejemplos:

```
    5        6
  -15        4
   20       17
 1340       26
```

Los números enteros se pueden representar en 8, 16 o 32 bits, e incluso 64 bits, y eso da origen a una escala de enteros cuyos rangos dependen de cada máquina

Enteros	-32768	*a*	32767
Enteros cortos	-128	*a*	127
Enteros largos	-2147483648	*a*	2147483647

Además de los modificadores `corto` y `largo` se pueden considerar `sin signo` (*unsigned*) y `con signo` (*signed*).

```
sin signo:    0 .. 65.5350
              0 .. 4294967296
```

Los enteros se denominan en ocasiones números de punto o coma fija. Los números enteros máximos y mínimos de una computadora[1] suelen ser -32.768 a +32.767. Los números enteros fuera de este rango no suelen representarse como enteros, sino como reales, aunque existen excepciones en los lenguajes de programación modernos como C, C++ y Java.

Reales: son el conjunto de números naturales, enteros, racionales e irracionales; siempre tienen un punto decimal y pueden ser positivos o negativos. Un número real consta de un entero y una parte decimal. Los siguientes ejemplos son números reales:

```
 0.08          3739.41
 3.7452         -52.321
-8.12             3.0
```

En aplicaciones científicas se requiere una representación especial para manejar números muy grandes, como la masa de la Tierra, o muy pequeños, como la masa de un electrón. Una computadora sólo puede representar un número fijo de dígitos. Este número puede variar de una máquina a otra, siendo ocho dígitos un número típico. Este límite provocará problemas para representar y almacenar números muy grandes o muy pequeños como son los ya citados o los siguientes:

```
4867213432    0.00000000387
```

Existe un tipo de representación denominado *notación exponencial* o *científica* y que se utiliza para números muy grandes o pequeños. Así,

```
367520100000000000000
```

se representa en notación científica descomponiéndolo en grupos de tres dígitos

```
367   520   100   000   000   000   000
```

y posteriormente en forma de potencias de 10

3.675201×10^{20}

y de modo similar

```
0.0000000000302579
```

se representa como

3.02579×10^{-11}

La representación en *coma flotante* es una generalización de notación científica. Observe que las siguientes expresiones son equivalentes:

$3.675201 \times 10^{19} = .3675207 \times 10^{20} = .03675201 \times 10^{21} = \ldots$
$= 36.75201 \times 10^{18} = 367.5201 \times 10^{17} = \ldots$

[1] En computadoras de 16 bits como IBM PC o compatibles.

Cristian Pilapanta 1006211-2202100232

En estas expresiones se considera la *mantisa* (parte decimal) al número real y el *exponente* (parte potencial) el de la potencia de diez.

```
36.75201    mantisa    18    exponente
```

Los tipos de datos reales se representan en coma o punto flotante y suelen ser de simple precisión, doble precisión o cuádruple precisión y suelen requerir 4 bytes, 8 bytes o 10-12 bytes, respectivamente. La tabla 3.2 muestra los datos reales típicos en compiladores C/C++.

Tabla 3.2. Tipos de datos reales (coma flotante) en el lenguaje C/C++.

Tipo	Rango de valores
real (*float*)	-3.4×10^{38} .. 3.4×10^{38}
doble (*double*)	-1.7×10^{-308} .. 1.7×10^{308}

3.5.2. Datos lógicos (*booleanos*)

El tipo *lógico* —también denominado *booleano*— es aquel dato que sólo puede tomar uno de dos valores:

`cierto` o `verdadero` (*true*) o `falso` (*false*).

Este tipo de datos se utiliza para representar las alternativas (*sí/no*) a determinadas condiciones. Por ejemplo, cuando se pide si un valor entero es par, la respuesta será verdadera o falsa, según sea par o impar.

C++ y Java soportan el tipo de dato *bool*.

3.5.3. Datos tipo carácter y tipo cadena

El tipo *carácter* es el conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato tipo carácter contiene un solo carácter. Los caracteres que reconocen las diferentes computadoras no son estándar; sin embargo, la mayoría reconoce los siguientes caracteres alfabéticos y numéricos:

- Caracteres alfabéticos (A, B, C, ..., Z) (a, b, c, ..., z).
- Caracteres numéricos (1, 2, ..., 9, 0).
- Caracteres especiales (+, -, *, /, ^, ..., :, <, >, $, ...).

Una *cadena* (*string*) de *caracteres* es una sucesión de caracteres que se encuentran delimitados por una comilla (apóstrofo) o comillas dobles, según el tipo de lenguaje de programación. La *longitud* de una cadena de caracteres es el número de ellos comprendido entre los separadores o limitadores. Algunos lenguajes tienen datos tipo *cadena*.

```
'Hola Mortimer'
'12 de octubre de 1492'
'Sr. McKoy'
```

3.6. Constantes y variables

Los programas de computadora contienen ciertos valores que no deben cambiar durante la ejecución del programa. Tales valores se llaman *constantes*. De igual forma, existen otros valores que cambiarán durante la ejecución del programa. A éstos se les llama *variables*. Una ***constante*** es un dato que permanece sin cambios durante todo el desarrollo del algoritmo o durante la ejecución del programa.

Constantes reales válidas

```
1.234
-0.1436
+ 54437324
```

Constantes reales no válidas

```
1234.63   (comas no permitidas)
82        (normalmente contienen un punto decimal, aunque
           existen lenguajes que lo admiten sin punto)
```

Constantes reales en notación científica

`3.374562E` equivale a 3.374562×10^2

Cristian Palpanta 1095211-2202100333

Una *constante tipo carácter* o *constante de caracteres* consiste en un carácter válido encerrado dentro de apóstrofos; por ejemplo:

```
'B'     '+'     '4'     ';'
```

Si se desea incluir el apóstrofo en la cadena, entonces debe aparecer como un par de apóstrofos, encerrados dentro de comillas simples.

```
" "
```

Una secuencia de caracteres se denomina normalmente una *cadena;* y una *constante tipo cadena* es una cadena encerrada entre apóstrofos. Por consiguiente:

```
'Juan Minguez'
```

y

```
'Pepe Luis Garcia'
```

son constantes de cadena válidas. Nuevamente, si un apóstrofo es uno de los caracteres en una constante de cadena, debe aparecer como un par de apóstrofos:

```
'John"s'
```

Constantes lógicas (booleanas)

Sólo existen dos constantes *lógicas* o *booleanas*:

```
verdadero     falso
```

La mayoría de los lenguajes de programación permiten diferentes tipos de constantes: *enteras, reales, caracteres* y *booleanas* o *lógicas,* y representan datos de esos tipos.

Una ***variable*** es un objeto o tipo de datos cuyo valor puede cambiar durante el desarrollo del algoritmo o ejecución del programa. Dependiendo del lenguaje, hay diferentes tipos de variables, tales como *enteras, reales, carácter, lógicas* y *de cadena*. Una variable que es de un cierto tipo puede tomar únicamente valores de ese tipo. Una variable de carácter, por ejemplo, puede tomar como valor sólo caracteres, mientras que una variable entera puede tomar sólo valores enteros.

Si se intenta asignar un valor de un tipo a una variable de otro tipo se producirá *un error de tipo*.

Una variable se identifica por los siguientes atributos: *nombre* que lo asigna y *tipo* que describe el uso de la variable.

Los nombres de las variables, a veces conocidos como ***identificadores***, suelen constar de varios caracteres alfanuméricos, de los cuales el primero normalmente es una letra. No se deben utilizar, aunque lo permita el lenguaje —caso de Fortran—, utilizar como nombres de identificadores palabras reservadas del lenguaje de programación. Nombres válidos de variables son:

```
A510
NOMBRES                 Letra           Salario Mes
NOTAS                   Horas           Segundo Apellido
NOMBRE_APELLIDOS[2]     Salario         Ciudad
```

Los nombres de las variables elegidas para el algoritmo o el programa deben ser significativos y tener relación con el objeto que representan; observe los casos siguientes:

```
NOMBRE    para representar nombres de personas.
PRECIOS   para representar los precios de diferentes artículos.
NOTAS     para representar las notas de una clase.
```

[2] Algunos lenguajes de programación admiten como válido el carácter subrayado en los identificadores.

Cristian Pulgarín 1095211-2202160234

Existen lenguajes —como Pascal— en los que es posible darle nombre a determinadas constantes típicas utilizadas en cálculos matemáticos, financieros y de otro tipo. Por ejemplo, las constantes $\pi = 3.141592...$ y $e = 2.718228$ (base de los logaritmos naturales) se les pueden dar los nombres `PI` y `E`.

```
Pi = 3.141592
E  = 2.718228
```

3.6.1. Declaración de constantes y variables

Normalmente los identificadores de las variables y de las constantes con nombre deben ser declaradas en los programas antes de ser utilizadas. La sintaxis de la declaración de una variable suele ser:

```
<tipo_de_dato> <nombre_variable> [=<expresión>]
```

EJEMPLO

```
car letra, abreviatura
ent numAlumnos = 25
real salario = 23.000
```

Si se desea dar un nombre (identificador) y un valor a una constante de modo que su valor no se pueda modificar posteriormente, su sintaxis puede ser así:

```
const <tipo_de_dato> <nombre_constante> =<expresión>
```

EJEMPLO

```
const doble PI = 3.141592
const cad nombre = 'Mackoy'
const car letra = 'c'
```

3.7. Expresiones y operadores

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales. Las mismas ideas son utilizadas en notación matemática tradicional; por ejemplo:

$$a + (b + 3) + \sqrt{c}$$

Aquí los paréntesis indican el orden de cálculo y $\sqrt{}$ representa la función raíz cuadrada.

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas. Una expresión consta de *operandos* y *operadores*. Según sea el tipo de objetos que manipulan, las expresiones se clasifican en:

- *Aritméticas.*
- *Relacionales.*
- *Lógicas.*
- *Carácter.*

El resultado de la expresión aritmética es de tipo numérico, el resultado de la expresión relacional y de una expresión lógica es de tipo lógico, el resultado de una expresión carácter es de tipo carácter.

Cristian Pilapanta 1095211-2202100035

3.7.1. Expresiones aritméticas

Las *expresiones aritméticas* son análogas a las fórmulas matemáticas. Las variables y constantes son numéricas (real o entera) y las operaciones son las aritméticas.

```
+              suma
-              resta
*              multiplicación
/              división
↑, **, ^       exponenciación
div, /         división entera
mod, %         módulo (resto)
```

Los símbolos +, -, *, ^ (↑ o **) y las palabras clave **div** y **mod** se conocen como *operadores aritméticos*. En la expresión

```
5 + 3
```

los valores 5 y 3 se denominan *operandos*. El valor de la expresión 5 + 3 se conoce como *resultado* de la expresión. Los operadores se utilizan de igual forma que en matemáticas. Por consiguiente, `A - B` se escribe en un algoritmo como `A * B` y `1/4 - C` como `C/4`. Al igual que en matemáticas el signo menos juega un doble papel, como resta en `A - B` y como negación en `-A`.

> No todos los operadores aritméticos existen en todos los lenguajes de programación; por ejemplo, en Fortran no existe **div** y **mod**. El operador de exponenciación es diferente según sea el tipo de lenguaje de programación elegido (^, ↑ en Basic, ** en Fortran).

Los cálculos que implican tipos de datos reales y enteros suelen dar normalmente resultados del mismo tipo si los operandos lo son también. Por ejemplo, el producto de operandos reales produce un valor real (tabla 3.3).

EJEMPLO

5×7	se representa por	`5 * 7`
$\frac{6}{4}$	se representa por	`6/4`
3^7	se representa por	`3^7`

Tabla 3.3. Operadores aritméticos.

Operador	Significado	Tipos de operandos	Tipo de resultado
–	Signo negativo	Entero o real	Entero o real
*	Multiplicación	Entero o real	Entero o real
/	División	Real	Real
div, /	División entera	Entero	Entero
mod, %	Módulo (resto)	Entero	Entero
++	Incremento	Entero	Entero
—	Decremento	Entero	Entero

Cristian Pilapanta 1095211-2202100036

Operadores *DIV* (/) y *MOD* (%)

El símbolo / se utiliza para la división real y la división entera el operador `div` (en algunos lenguajes, por ejemplo, Basic, se suele utilizar el símbolo \ que representa la división entera). El operador `mod` representa el resto de la división entera, y la mayoría de los lenguajes utilizan el símbolo %.

```
A div B
```

Sólo se puede utilizar si `A` y `B` son expresiones enteras y obtiene la parte entera de `A/B`. Por consiguiente,

```
19 div 6   19/6
```

toma el valor 3. Otro ejemplo puede ser la división `15/6`

```
15          |6
 3          2    cociente
|
resto
```

En forma de operadores resultará la operación anterior

```
15 div 6 = 2        15 mod 6 = 3
```

Otros ejemplos son:

```
19 div 3   equivale a 6
19 mod 6   equivale a 1
```

Cristian Pilapanta 1095211-2202100337

EJEMPLO 3.1

Los siguientes ejemplos muestran resultados de expresiones aritméticas:

expresión	resultado	expresión	resultado
10.5/3.0	3.5	10/3	3
1/4	0.25	18/2	9
2.0/4.0	0.5	30/30	1
6/1	6.0	6/8	0
30/30	1.0	10%3	1
6/8	0.75	10%2	0

Operadores de incremento y decremento

Los lenguajes de programación C/C++, Java y C# soportan los operadores unitarios (unarios) de incremento, `++`, y decremento, `--`. El operador de incremento (++) aumenta el valor de su operando en una unidad, y el operador de decremento (--) disminuye también en una unidad. El valor resultante dependerá de que el operador se emplee como prefijo o como sufijo (antes o después de la variable). Si actúa como prefijo, el operador cambia el valor de la variable y devuelve este nuevo valor; en caso contrario, si actúa como sufijo, el resultado de la expresión es el valor de la variable, y después se modifica ésta.

`++i`	Incrementa `i` en 1 y después utiliza el valor de `i` en la correspondiente expresión.
`i++`	Utiliza el valor de `i` en la expresión en que se encuentra y después se incrementa en 1.
`--i`	Decrementa `i` en 1 y después utiliza el nuevo valor de `i` en la correspondiente expresión.
`i-i--`	Utiliza el valor de `i` en la expresión en que se encuentra y después se incrementa en 1.

EJEMPLO

```
n = 5
escribir n
escribir n++
escribir n
n = 5
escribir n
escribir ++n
escribir n
```

Al ejecutarse el algoritmo se obtendría:

```
5
5
6
5
6
6
```

Cristian Palapanta 1006211-2202100238

3.7.2. Reglas de prioridad

Las expresiones que tienen dos o más operandos requieren unas reglas matemáticas que permitan determinar el orden de las operaciones, éstas se denominan reglas de *prioridad* o *precedencia* y son:

1. Las operaciones que están encerradas entre paréntesis se evalúan primero. Si existen diferentes paréntesis anidados (interiores unos a otros), las expresiones más internas se evalúan primero.
2. Las operaciones aritméticas dentro de una expresión suelen seguir el siguiente orden de prioridad:
   - operador ( )
   - operadores ++, – – + y – unitarios
   - operadores *, /, % (producto, división, módulo)
   - operadores +, – (suma y resta).

En los lenguajes que soportan la operación de exponenciación, este operador tiene la mayor prioridad.

En caso de coincidir varios operadores de igual prioridad en una expresión o subexpresión encerrada entre paréntesis, el orden de prioridad en este caso es de izquierda a derecha, y a esta propiedad se denomina *asociatividad*.

EJEMPLO 3.2

¿Cuál es el resultado de las siguientes expresiones?

a) 3 + 6 * 14

b) 8 + 7 * 3 + 4 * 6

Solución

a) 3 + 6 * 14

3 + 84

87

b) 8 + 7 * 3 + 4 * 6

8 + 21 24

29 + 24

53

EJEMPLO 3.3

Obtener los resultados de las expresiones:

```
-4 * 7 + 2 ^ 3 / 4 - 5
```

Solución

```
-4 * 7 + 2 ^ 3 / 4 - 5
```

resulta

```
-4 * 7 + 8 / 4 - 5
-28 + 8 / 4 - 5
-28 + 2 - 5
-26 - 5
-31
```

EJEMPLO 3.4

Convertir en expresiones aritméticas algorítmicas las siguientes expresiones algebraicas:

$5 \cdot (x + y)$

$\dfrac{x + y}{u + \dfrac{w}{a}}$

$a^2 + b^2$

$\dfrac{x}{y} \cdot (z + w)$

Los resultados serán:

```
5 *(x + y)
a ^2 + b ^2
(x + y) / (u + w/a)
x / y *(z + w)
```

Cristian Pilapanta 1086211-2202100340

1086211220210040

EJEMPLO 3.5

Los paréntesis tienen prioridad sobre el resto de las operaciones:

`A * (B + 3)`	la constante 3 se suma primero al valor de `B`, después este resultado se multiplica por el valor de `A`.
`(A * B) + 3`	`A` y `B` se multiplican primero y a continuación se suma 3.
`A + (B + C) + D`	esta expresión equivale a `A + B + C + D`
`(A + B/C) + D`	equivale a `A + B/C + D`
`A * B/C * D`	equivale a `((A * B)/C) * D` y no a `(A * B)/(C * D)`.

EJEMPLO 3.6

Evaluar la expresión `12 + 3 * 7 + 5 * 4`.

En este ejemplo existen dos operadores de igual prioridad, * (multiplicación); por ello los pasos sucesivos son:

```
12 + 3 * 7 + 5 * 4
     ------
       21
```

```
12 + 21 + 5 * 4
         ↓
         20
12 + 21 + 20 = 53
```

3.7.3. Expresiones lógicas (*booleanas*)

Un segundo tipo de expresiones es la *expresión lógica* o *booleana*, cuyo valor es siempre verdadero o falso. Recuerde que existen dos constantes lógicas, *verdadera* (*true*) y *falsa* (*false*) y que las variables lógicas pueden tomar sólo estos dos valores. En esencia, una *expresión lógica* es una expresión que sólo puede tomar estos dos valores, *verdadero* y *falso*. Se denominan también *expresiones booleanas* en honor del matemático británico George Boole, que desarrolló el Álgebra lógica de Boole.

Las expresiones lógicas se forman combinando constantes lógicas, variables y otras expresiones lógicas, utilizando los *operadores lógicos* `not`, `and` y `or` y los *operadores relacionales* (de relación o comparación) =, <, >, < =, > =, < >.

Operadores de relación

Este tipo de operadores, conocidos también como operadores relacionales, permiten realizar comparaciones de valores de tipo numérico o carácter. Además, sirven para expresar las condiciones en los algoritmos. Los operadores de relación se muestran en la tabla 3.4. El formato general para las comparaciones es:

```
expresión 1    operador de relación    expresión 2
```

y el resultado de la operación será verdadero o falso. Así, por ejemplo, si `A = 4` y `B = 3`, entonces

`A > B` *es verdadero*

Tabla 3.4. Operadores de relación.

Operador	Significado
<	menor que
>	mayor que
=, ==	igual que
<=	menor o igual que
>=	mayor o igual que
<>, !=	distinto de

mientras que:

`(A - 2) < (B - 4)` *es falso.*

Los operadores de relación se pueden aplicar a cualquiera de los cuatro tipos de datos estándar: *enteros, real, lógico, carácter*. La aplicación a valores numéricos es evidente. Los ejemplos siguientes son significativos:

N1	N2	Expresión lógica	Resultado
3	6	3 < 6	verdadero
0	1	0 > 1	falso
4	2	4 = 2	falso
8	5	8 <= 5	falso
9	9	9 >= 9	verdadero
5	5	5 <> 5	falso

Para realizar comparaciones de datos tipo carácter se requiere una secuencia de ordenación de los caracteres similar al orden creciente o decreciente. Esta ordenación suele ser alfabética, tanto mayúsculas como minúsculas, y numé-

Cristian Pilapanta 1096211-2202100040

rica, considerándolas de modo independiente. Pero si se consideran caracteres mixtos, se debe recurrir a un código normalizado como ASCII (apéndice A). Aunque no todas las computadoras siguen el código normalizado en su juego completo de caracteres, sí son prácticamente estándar los códigos de los caracteres alfanuméricos más usuales. Estos códigos normalizados son:

- Los caracteres especiales #, %, $, (,), +, –, /, . . . , exigen la consulta del código de ordenación.
- Los valores de los caracteres que representan a los dígitos están en su orden natural. Esto es, '0'<'1', '1'<'2', . . . , '8'<'9'.
- Las letras mayúsculas A a Z siguen el orden alfabético ('A'<'B', 'C'<'F', etcétera).
- Si existen letras minúsculas, éstas siguen el mismo criterio alfabético ('a'<'b', 'c'<'h', y así sucesivamente).

En general, los cuatro grupos anteriores están situados en el código ASCII en orden creciente. Así, '1'<'A' y 'B'<'C'. Sin embargo, para tener completa seguridad será preciso consultar el código de caracteres de su computadora (normalmente, el **ASCII**, American Standard Code for Information Interchange o bien el ambiguo código EBCDIC, Extended Binary-Coded Decimal Interchange Code, utilizado en computadoras IBM diferentes a los modelos PC).

Cuando se utilizan los operadores de relación, con valores lógicos, la constante *false (falsa)* es menor que la constante *true (verdadera)*.

```
false < true
true > false
```

Si se utilizan los operadores relacionales = y <> para comparar cantidades numéricas es importante recordar que la mayoría de los *valores reales no pueden ser almacenados exactamente*. En consecuencia, las expresiones lógicas formales con comparación de cantidades reales con (=), a veces se evalúan como falsas, incluso aunque estas cantidades sean algebraicamente iguales. Así:

```
(1.0 / 3.0) * 3.0 = 1.0
```

teóricamente es verdadera y, sin embargo, al realizar el cálculo en una computadora se puede obtener `0.999999...` y, en consecuencia, el resultado es falso; esto es debido a la precisión limitada de la aritmética real en las computadoras. Por consiguiente, a veces *deberá excluir las comparaciones con datos de tipo real*.

Operadores lógicos

Los *operadores lógicos* o *booleanos* básicos son `not` (*no*), `and` (*y*) y `or` (*o*). La tabla 3.5 recoge el funcionamiento de dichos operadores.

Tabla 3.5. Operadores lógicos.

Operador lógico	Expresión lógica	Significado
`no` (*not*), `!`	`no` p (*not* p)	negación de *p*
`y` (*and*), `&&`	p `y` q (p *and* q)	conjunción de *p* y *q*
`o` (*o*), `||`	p `o` q (p *o* q)	disyunción de *p* y *q*

Las definiciones de las operaciones `no`, `y`, `o` se resumen en unas tablas conocidas como *tablas de verdad*.

a	no a	
verdadero	falso	no (6>10) es verdadera
falso	verdadero	ya que (6>10) es falsa.

Cristian Pilquimán 1095211-2202100241

a	b	a y b	
verdadero	verdadero	verdadero	a y b es verdadera sólo
verdadero	falso	falso	si a y b son verdaderas.
falso	verdadero	falso	
falso	falso	falso	

a	b	a y b	
verdadero	verdadero	verdadero	a o b es verdadera cuando
verdadero	falso	verdadero	a, b o ambas son verdaderas.
falso	verdadero	verdadero	
falso	falso	falso	

En las expresiones lógicas se pueden mezclar operadores de relación y lógicos. Así, por ejemplo:

```
(1 < 5) y (5 < 10)          es verdadera
(5 > 10) o ('A' < 'B')      es verdadera, ya que 'A' < 'B'
```

EJEMPLO 3.7

La tabla 3.6 resume una serie de aplicaciones de expresiones lógicas.

Tabla 3.6. Aplicaciones de expresiones lógicas.

Expresión lógica	Resultado	Observaciones
`(1 > 0) y (3 = 3)`	verdadero	
`no PRUEBA`	verdadero	`PRUEBA` es un valor lógico falso.
`(0 < 5) o (0 > 5)`	verdadero	
`(5 <= 7) y (2 > 4)`	falso	
`no (5 <> 5)`	verdadero	
`(número = 1) o (7 >= 4)`	verdadero	`número` es una variable entera de valor 5.

Cristian Pilapanta 1006211-2202100046

Prioridad de los operadores lógicos

Los operadores aritméticos seguían un orden específico de prioridad cuando existía más de un operador en las expresiones. De modo similar, los operadores lógicos y relaciones tienen un orden de prioridad (tablas 3.7 y 3.8).

Tabla 3.7. Prioridad de operadores (lenguaje Pascal).

Operador	Prioridad
`no` *(not)*	más alta (primera ejecutada).
`/`, `*`, `div`, `mod`, `y` *(and)*	↓
`+`, `-`, `o` *(or)*	↓
`<`, `>`, `=`, `<=`, `>=`, `<>`	más baja (última ejecutada).

Tabla 3.8. Prioridad de operadores (lenguajes C, C++, C# y Java).

Operador	Prioridad
`++` y `--` (incremento y decremento en 1), `+`, `-`, `!`	más alta
`*`, `/`, `%` (módulo de la división entera)	↓
`+`, `-`, (suma, resta)	↓
`<`, `<=`, `>`, `>=`	↓
`==` (igual a), `!=` (no igual a)	↓
`&&` (y lógica, AND)	↓
`||` (o lógica, or)	↓
`=`, `+=`, `-=`, `*=`, `/=`, `%=` (operadores de asignación)	más baja

Al igual que en las expresiones aritméticas, los paréntesis se pueden utilizar y tendrán prioridad sobre cualquier operación.

EJEMPLO 3.8

```
no 4 > 6                    produce un error, ya que el operador no se aplica a 4
no (4 > 14)                 produce un valor verdadero
(1.0 < x) y (x < z + 7.0)   si x vale 7 y z vale 4, se obtiene un valor verdadero
```

3.7.4. Reglas generales de prioridad y asociatividad

La **prioridad** o **precedencia** determina el orden en que se aplican los operadores a un valor. Los operadores C vienen en una tabla con quince grupos; los operadores del grupo 1 tienen mayor prioridad que los del grupo 2, y así sucesivamente (tabla 3.9):

- Si dos operadores se aplican al mismo operando, el que tiene mayor prioridad se aplica primero.
- Todos los operadores del mismo grupo tienen igual prioridad y asociatividad.
- La **asociatividad** izquierda-derecha significa aplicar el operador más a la izquierda primero, y en la asociatividad derecha-izquierda se aplica primero el operador más a la derecha.
- Los paréntesis tienen la máxima prioridad.

Tabla 3.9. Prioridad de operadores.

Prioridad	Operadores	Asociatividad
1	`-> [ ] ( )`	I – D
2	`++ -- ~ ! - + & * sizeof (tipo)`	D – I
3	`* / %`	I – D
4	`+ -`	I – D
5	`<< >>`	I – D
6	`< <= > >=`	I – D
7	`== !=`	I – D
8	`&`	I – D
9	`^`	I – D
10	`| &&`	I – D
11	`||`	I – D
12	`?: (expresión condicional)`	I – D
13	`= *= /= %= += -=`	D – I
14	`<<= >>= &= /= ^=`	D – I
15	`, (operador coma)`	I – D

I-D: *Izquierda-Derecha.*
D-I: *Derecha-Izquierda.*

3.8. Funciones internas

Las operaciones que se requieren en los programas exigen (en numerosas ocasiones), además de las operaciones aritméticas básicas, ya tratadas, un número determinado de operadores especiales que se denominan *funciones internas*, incorporadas o estándar. Por ejemplo, la función `ln` se puede utilizar para determinar el logaritmo neperiano de un número y la función `raiz2 (sqrt)` calcula la raíz cuadrada de un número positivo. Existen otras funciones que se utilizan para determinar las funciones trigonométricas.

Cristian Pilapanta 1095211-2202100047

La tabla 3.10 recoge las funciones internas más usuales, siendo x el argumento de la función.

Tabla 3.10. Funciones internas.

Función	Descripción	Tipo de argumento	Resultado
`abs(x)`	valor absoluto de x	entero o real	igual que argumento
`arctan(x)`	arco tangente de x	entero o real	real
`cos(x)`	coseno de x	entero o real	real
`exp(x)`	exponencial de x	entero o real	real
`ln(x)`	logaritmo neperiano de x	entero o real	real
`log10(x)`	logaritmo decimal de x	entero o real	real
`redondeo(x)` (*round*(x))*	redondeo de x	real	entero
`seno(x)` (*sin*(x))*	seno de x	entero o real	real
`cuadrado(x)` (*sqr*(x))*	cuadrado de x	entero o real	igual que argumento
`raiz2(x)` (*sqrt*(x))*	raíz cuadrada de x	entero o real	real
`trunc(x)`	truncamiento de x	real	entero

* Terminología en inglés.

EJEMPLO 3.9

Las funciones aceptan argumentos reales o enteros y sus resultados dependen de la tarea que realice la función:

Expresión	Resultado
`raiz2 (25)`	5
`redondeo (6.5)`	7
`redondeo (3.1)`	3
`redondeo (-3.2)`	-3
`trunc (5.6)`	5
`trunc (3.1)`	3
`trunc (-3.8)`	-3
`cuadrado (4)`	16
`abs (9)`	9
`abs (-12)`	12

EJEMPLO 3.10

Utilizar las funciones internas para obtener la solución de la ecuación cuadrática $ax^2 + bx + c = 0$. Las raíces de la ecuación son:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

o lo que es igual:

$$x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Cristian Pilapanta 1095211-2202100048

Las expresiones se escriben como

```
x1 = (-b + raiz2 (cuadrado (b) - 4 * a * c)) / (2 * a)
x2 = (-b - raiz2 (cuadrado (b) - 4 * a * c)) / (2 * a)
```

Si el valor de la expresión

```
raiz2 (cuadrado (b) - 4 * a * c)
```

es negativo se producirá un error, ya que la raíz cuadrada de un número negativo no está definida.

3.8.1. Funciones matemáticas de Java

La clase `Math` contiene una colección de funciones matemáticas que se pueden necesitar, dependiendo del tipo de programación que desee realizar. Por ejemplo, para calcular x^n

```
Math.pow(x,n)
```

Sin embargo, para calcular x^2 puede ser más eficiente calcular `x*x`. Para extraer la raíz cuadrada de un número se puede utilizar el método `sqrt`:

```
double x = 9.0;
double y = Math.sqrt(x);
System.out.println("y = " + y); // se visualiza 3
```

EJEMPLO

Calcular el valor de $(-b + \sqrt{(b^2 - 4ac)})/2a$ `(-b + Math.sqrt(b*b -4*a*c))/(2*a)`
La tabla 3.11 muestra los métodos matemáticos de la clase `Math`.

Tabla 3.11. Métodos matemáticos.

Función	Devuelve
`Math.sqrt(x)`	raíz cuadrada de x (x ≥ 0)
`Math.pow(x,y)`	x^y
`Math.exp(x)`	e^x
`Math.log(x)`	logaritmo natural (ln(x), x > 0)
`Math.round(x)`	entero más próximo a x (long)
`Math.ceil(x)`	entero más pequeño ≥ x (double)
`Math.floor(x)`	entero más próximo ≤ x (double)
`Math.abs(x)`	valor absoluto de x
`Math.max(x,y)`	valor mayor de x y y
`Math.min(x,y)`	valor menor de x y y
`Math.sin(x)`	seno de x (x en radianes)
`Math.cos(x)`	coseno de x (x en radianes)
`Math.tan(x)`	tangente de x (x en radianes)
`Math.asin(x)`	arco seno de x
`Math.acos(x)`	arco coseno de x
`Math.atan(x)`	arco tangente de x
`Math.atan2(y,x)`	arco cuya tangente es y/x
`Math.toRadianes(x)`	convierte x grados a radianes
`Math.toDegrees(x)`	convierte x radianes a grados

Cristian Pilapanta 1005211-2202100048

3.9. Operación de asignación

La operación de asignación es el modo de almacenar valores a una variable; se representa con el símbolo u operador ← (en la mayoría de los lenguajes de programación, como C, C++, Java, el signo de la operación asignación es =). Esta operación también se conoce como *instrucción* o *sentencia* de asignación cuando se refiere a un lenguaje de programación, el formato general de dicha operación es:

```
<nombre de la variable> ← <expresión>
```

expresión es igual a `expresión`, `variable` o `constante`

La flecha (operador de asignación) se sustituye en otros lenguajes por = (Visual Basic, Fortran), := (Pascal) o = (Java, C++, C#). Sin embargo, es preferible el uso de la flecha en la redacción del algoritmo para evitar ambigüedades, dejando el uso del símbolo = exclusivamente para el operador de igualdad.

La operación de asignación:

```
A ← 5
```

significa que a la variable `A` se le ha asignado el valor 5.

La acción de asignar es *destructiva*, ya que el valor que tuviera la variable antes de la asignación se pierde y se reemplaza por el nuevo valor. Así, en la secuencia de operaciones

```
A ← 25
A ← 134
A ← 5
```

cuando éstas se ejecutan, el último valor que toma `A` será 5 (los valores 25 y 134 han desaparecido).

La computadora ejecuta la sentencia de asignación en dos pasos. En el primero de ellos, el valor de la expresión al lado derecho del operador se calcula, obteniéndose un valor de un tipo específico. En el segundo caso, este valor se almacena en la variable cuyo nombre aparece a la izquierda del operador de asignación, sustituyendo al valor que tenía anteriormente.

```
X ← Y + 2
```

el valor de la expresión `Y + 2` se asigna a la variable `X`.

Es posible utilizar el mismo nombre de variable en ambos lados del operador de asignación. Por ello, acciones como:

```
N ← N + 1
```

tienen sentido. Se determina el valor actual de la variable `N`, se incrementa en 1 y a continuación el resultado se asigna a la misma variable `N`. Sin embargo, desde el punto de vista matemático no tiene sentido `N ← N + 1`.

Las acciones de asignación se clasifican según sea el tipo de expresiones en: *aritméticas, lógicas* y *de caracteres.*

3.9.1. Asignación aritmética

Las expresiones en las operaciones de asignación son aritméticas:

```
AMN ← 3 + 14 + 8    se evalúa la expresión 3 + 14 + 8 y se asigna a la variable AMN, es decir, 25 será el valor que
                    toma AMN
TER1 ← 14.5 + 8
TER2 ← 0.75 * 3.4 COCIENTE ← TER1/TER2
```

Se evalúan las expresiones `14.5 + 8` y `0.75 * 3.4` y en la tercera acción se dividen los resultados de cada expresión y se asigna a la variable `COCIENTE`, es decir, las tres operaciones equivalen a `COCIENTE ← (14.5 + 8)/(0.75 * 3.4)`.

Cristian Pilapanta 1096211-2200100049

Otro ejemplo donde se pueden comprender las modificaciones de los valores almacenados en una variable es el siguiente:

```
A ← 0          la variable A toma el valor 0
N ← 0          la variable N toma el valor 0
A ← N + 1      la variable A toma el valor 0 + 1, es decir 1.
```

El ejemplo anterior se puede modificar para considerar la misma variable en ambos lados del operador de asignación:

```
N ← 2
N ← N + 1
```

En la primera acción N toma el valor 2 y en la segunda se evalúa la expresión N + 1, que tomará el valor 2 + = 3 y se asignará nuevamente a N, que tomará el valor 3.

3.9.2. Asignación lógica

La expresión que se evalúa en la operación de asignación es lógica. Supóngase que M, N y P son variables de tipo lógico.

```
M ← 8 < 5
N ← M o (7 <= 12)
P ← 7 > 6
```

Tras evaluar las operaciones anteriores, las variables M, N y P tomarán los valores *falso, verdadero, verdadero.*

Cristian Pilapanta 1065211-2202100560

3.9.3. Asignación de cadenas de caracteres

La expresión que se evalúa es de tipo cadena:

```
x ← '12 de octubre de 1942'
```

La acción de asignación anterior asigna la cadena de caracteres '12 de octubre de 1942' a la variable tipo cadena x.

3.9.4. Asignación múltiple

Todos los lenguajes modernos admiten asignaciones múltiples y con combinaciones de operadores, además de la asignación única con el operador ←. Así, se puede usar el operador de asignación (←) precedido por cualquiera de los siguientes operadores aritméticos: +, –, *, /, % (tabla 3.12). La sintaxis es la siguiente:

```
<nombre_variable> ← <variable> <operador> <expresión>
```

y es equivalente a:

```
variable operador ← expresión
```

EJEMPLO

```
c ← c + 5          equivale a          c +← 5
a ← a * (b + c)    equivale a          a *← b + c
```

si lo prefiere, puede utilizar el signo de asignación (=) de C, C++, Java o C#.

Caso especial

Los lenguajes C, C++, Java y C# permiten realizar múltiples asignaciones en una sola sentencia:

```
a = b = c = d = e = n +35;
```

Tabla 3.12. Operadores aritméticos de asignación múltiple.

Operador de asignación	Ejemplo	Operación	Resultado
Entero a = 3, b = 5, c = 4, d = 6, e = 10			
* =	a + = 8	a = a + 8	a = 11
- =	b - = 5	b = b - 5	b = 0
* =	c * = 4	c = c * 4	c = 16
/ =	d / = 3	d = d / 3	d = 2
% =	e % = 9	e = e % 9	e = 1

3.9.5. Conversión de tipo

En las asignaciones no se pueden asignar valores a una variable de un tipo incompatible al suyo. Se presentará un error si se trata de asignar valores de tipo carácter a una variable numérica o un valor numérico a una variable tipo carácter.

Cristian Pilapanta 1095211-2202100651

1095211222100651

EJEMPLO 3.11

¿Cuáles son los valores de A, B *y* C *después de la ejecución de las siguientes operaciones?*

```
A ← 3
B ← 4
C ← A + 2 * B
C ← C + B
B ← C - A
A ← B * C
```

En las dos primeras acciones A y B toman los valores 3 y 4.

```
C ← A + 2 * B        la expresión A + 2 * B tomará el valor 3 + 2 * 4 = 3 + 8 = 11
C ← 11
```

La siguiente acción

```
C ← C + B
```

producirá un valor de 11 + 4 = 15

```
C ← 15
```

En la acción B ← C - A se obtiene para B el valor 15 - 3 = 12 y por último:

```
A ← B * C
```

A tomará el valor B * C, es decir, 12 * 15 = 180; por consiguiente, el último valor que toma A será 180.

EJEMPLO 3.12

¿Cuál es el valor de x después de las siguientes operaciones?

```
x ← 2
x ← cuadrado(x + x)
x ← raiz2(x + raiz2(x) + 5)
```

Los resultados de cada expresión son:

```
x ← 2                          x toma el valor 2
x ← cuadrado(2 + 2)            x toma el valor 4 al cuadrado; es decir 16
x ← raiz2(16 + raiz2(16) + 5)
```

en esta expresión se evalúa primero `raiz2(16)`, que produce 4 y, por último, `raiz2(16+4+5)` proporciona `raiz2(25)`, es decir, 5. Los resultados de las expresiones sucesivas anteriores son:

```
x ← 2
x ← 16
x ← 5
```

3.10. Operadores avanzados

Los lenguajes de programación modernos orientados a objetos e interpretados soportan unos operadores especiales adaptados a sus características específicas y que potencian sus reglas de sintaxis y de escritura de programas: operador condicional, operador coma, operadores especiales, operadores de manipulación de bits y operador `sizeof`, entre otros.

Cristian Pilapanta 1095211-2202100652

3.10.1. Operador condicional ?: (C/C++, Java)

El **operador condicional ?:** es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Tiene asociatividad a derechas (derecha a izquierda). Al ser un operador ternario requiere tres operandos. El operador *condicional* se utiliza para reemplazar a la sentencia `if-else` lógica en algunas expresiones. El formato del operador condicional es:

```
expresión_c ? expresión_v : expresión_f;
```

Se evalúa `expresión_c` y su valor (cero = *falso*, distinto de cero = *verdadero*) determina cuál es la expresión a ejecutar; si la condición es verdadera se ejecuta `expresión_v` y si es falsa se ejecuta `expresión_f`.

La figura 3.5 muestra el funcionamiento del operador condicional. Otros ejemplos del uso del operador ?: son:

```
n >= 0 ? 1 : -1 → /* 1 si n es positivo, -1 si es negativo */ m >= n ? m : n
                → /* devuelve el mayor valor de m y n */
```

Escribe x, y escribe el carácter fin de línea `(\n)` si x%5(resto 5) es 0, caso contrario un tabulador`(\t)`.

printf `("%d %c", x, x%5 ?'\t':'\n')`.

```
(ventas >150 000) ?     comisión = 100    :     comisión = 0;

si ventas es mayor      si ventas no es
que 150 000 se          mayor que 150 000 se
ejecuta:                ejecuta:
comisión = 100          comisión = 100
```

Figura 3.5. Formato de un operador condicional.

La precedencia de ?: es menor que la de cualquier otro operando tratado hasta ese momento. Su asociatividad es a derechas.

3.10.2. Operador coma

El *operador coma* permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes, de izquierda a derecha. La expresión más a la derecha determina el resultado global. El uso del operador coma es como sigue:

```
expresión¹, expresión², expresión³, ..., expresión^n
```

Cada expresión se evalúa comenzando desde la izquierda y continuando hacia la derecha. Por ejemplo, en:

```
int i = 10, j = 25;
```

dado que el operador coma se asocia de izquierda a derecha, la primera variable `i` está declarada e inicializada antes que la segunda variable j. Otros ejemplos son:

```
i++, j++ ;          equivale a      i++; j++;
i++, j++, k++ ;     equivale a      i++; j++; k++;
```

El operador coma tiene la menor prioridad de todos los operadores C y se asocia de izquierda a derecha.

El resultado de la expresión global se determina por el valor de *expresión*. Por ejemplo,

```
int i, j, resultado;
resultado = j = 10, i = j, ++i;
```

El valor de esta expresión es 11. En primer lugar, a j se le asigna el valor 10, a continuación a `i` se le asigna el valor de j. Por último, `i` se incrementa en 1 y pasa a ser 11, que es el resultado de la expresión. La técnica del operador coma permite operaciones interesantes:

```
i = 10;
j = (i = 12, i + 8);
```

Cuando se ejecute la sección de código anterior, j vale 20, ya que `i` vale 10 en la primera sentencia, en la segunda toma `i` el valor 12 y al sumar `i + 8` resulta 20.

3.10.3. Operadores especiales: ( ), [ ]

Algunos operadores especiales que sirven para propósitos diferentes son admitidos por C. Entre ellos destacan: `( )`,`[ ]`.

El operador ( )

El operador `( )` es el operador de llamada a funciones. Sirve para encerrar los argumentos de una función, efectuar conversiones explícitas de tipo, indicar en el seno de una declaración que un identificador corresponde a una función y resolver los conflictos de prioridad entre operadores.

El operador [ ]

Sirve para dimensionar los arreglos (*arrays*) y designar un elemento de un arreglo (*array*).

Cristian Palpanta 1085211-2202100257

Ejemplos de ello:

```
double v[20];                /* define un arreglo (array) de 20 elementos */ printf("v[2] =
%e",v[2]);                   /* escribe el elemento 2 de v */
return v[i-INFERIOR];        /* devuelve el elemento i-INFERIOR */
```

3.10.4. Operador `sizeof`

Con frecuencia su programa necesita conocer el tamaño en bytes de un tipo de dato o variable. C proporciona el *operador* `sizeof`, que toma un argumento, bien un tipo de dato o bien el nombre de una variable (escalar, arreglo [*array*], registro, etcétera). El formato del operador es:

```
sizeof (nombre_variable) sizeof nombre_variable sizeof (tipo_dato) sizeof (expresión)
```

EJEMPLO 3.13

Si se supone que el tipo `int` *consta de cuatro bytes y el tipo* `double` *consta de ocho bytes, las siguientes expresiones proporcionarán los valores 1, 4 y 8, respectivamente:*

```
sizeof(char)
#
sizeof(unsigned int) sizeof(double)
```

El operador `sizeof` puede aplicarse también a expresiones y se puede escribir de la siguiente manera:

```
printf("La variable k es %d bytes", sizeof(k));
printf("La expresión a + b ocupa %d bytes",sizeof (a + b));
```

El operador `sizeof` es un operador unitario, ya que opera sobre un valor único. Este operador produce un resultado que es el tamaño, en bytes, del dato o tipo de dato especificado. Debido a que la mayoría de los tipos de datos y variables requieren diferentes cantidades de almacenamiento interno de acuerdo con cada computadora, el operador `sizeof` permite consistencia de programas en distintos tipos de computadoras.

El operador `sizeof` se denomina también *operador en tiempo de compilación*, ya que, en tiempo de compilación, el compilador sustituye cada ocurrencia de `sizeof` en su programa por un valor entero sin signo (`unsigned`). El operador `sizeof` se utiliza en programación avanzada.

EJEMPLO 3.14

Suponga que desea conocer el tamaño, en bytes, de variables de coma flotante y de doble precisión de su computadora. El siguiente programa realiza esa tarea:

```
* Imprime el tamaño de valores de coma flotante y double */ #include <stdio.h>
int main()
{
   printf("El tamaño de variables de coma flotante es %d\n", sizeof(float));
   printf("El tamaño de variables de doble precisión es %d\n", sizeof(double));
   return 0;
}
```

Este programa producirá diferentes resultados en distintas clases de computadoras. Compilando este programa bajo C, el programa produce la salida siguiente:

```
El tamaño de variables de coma flotante es: 4; mientras que el tamaño de variables de doble
precisión es: 8.
```

Cristian Pilapanta 1006211-2202100256

3.10.5. Operadores de manipulación de bits (*bitwise*, Java)

Cuando se trabaja con tipos enteros, existen operadores que pueden trabajar directamente con los bits que forman los enteros. Esto significa que se pueden utilizar técnicas de enmascaramiento para obtener bits individuales de un número.

Los operadores de manipulación de bits son:

```
& ("and")   | ("or")        ^ ("xor")       ~ ("not")
```

estos operadores trabajan sobre patrones de bits.

EJEMPLO

```
int n;
int cuartoBitMenorPeso = (n & 8);
```

producirá un 1 si el cuarto bit, empezando por la derecha (el menos significativo), de n es 1, y 0 si no lo es.

```
xxxx1xxx          xxxx0xxx
&    1000         &    1xxx
------            ------
   1                 0
```

Los operadores >> y << desplazan un patrón de bits a la derecha o a la izquierda. Estos operadores son muy útiles cuando se necesita construir patrones de bits para enmascarar bits.

El operador >>> rellena los bits superiores con cero, mientras que >> amplía el bit de signo en los bits de mayor peso.

3.10.6. Prioridad y asociatividad

La prioridad o precedencia de operadores determina el orden en el que se aplican los operadores a un valor. Los operadores de C se presentan en la tabla 3.13 en quince grupos. Los operadores del grupo 1 tienen mayor prioridad que los del grupo 2, y así sucesivamente:

- Si dos operadores se aplican al mismo operando, el operador con mayor prioridad se aplica primero.
- Todos los operadores del mismo grupo tienen igual prioridad y asociatividad.
- La asociatividad izquierda-derecha significa aplicar el operador más a la izquierda primero, y en la asociatividad derecha-izquierda se aplica primero el operador más a la derecha.
- Los paréntesis tienen la máxima prioridad.

En general, el orden de evaluación de primero a último es:

- Operadores de incremento y decremento.
- Operadores aritméticos.
- Comparaciones.
- Operadores lógicos.
- Expresiones de asignación.

Cristian Pulgarín 1095211-2202100569

Tabla 3.13. Prioridad de operadores.

Prioridad	Operadores	Asociatividad
1	`-> [ ] ( )`	I – D
2	`++ -- ~ ! - + & * sizeof (tipo)`	D – I
3	`* / %`	I – D
4	`+ -`	I – D
5	`<<>>`	I – D
6	`< <= > >=`	I – D
7	`== !=`	I – D
8	`&`	I – D
9	`^`	I – D
10	`| &&`	I – D
11	`||`	I – D
12	`?: (expresión condicional)`	I – D
13	`= *= /= %= += -=`	D – I
14	`<<= >>= &= /= ^=`	D – I
15	`, (operador coma)`	I – D

I-D: *Izquierda-Derecha.*
D-I: *Derecha-Izquierda.*

Cristian Pilapanta 1095211-2202100101

3.10.7. Conversiones de tipos

Con frecuencia, se necesita convertir un valor de un tipo a otro sin cambiar el valor que representa. Las *conversiones de tipos* pueden ser *implícitas* (ejecutadas automáticamente) o *explícitas* (solicitadas de manera específica por el programador). C hace automáticamente muchas conversiones de tipos:

- C convierte valores cuando se asigna un valor de un tipo a una variable de otro tipo.
- C convierte valores cuando se combinan tipos mixtos en expresiones.
- C convierte valores cuando se pasan argumentos a funciones.

Conversión implícita

Los tipos fundamentales (básicos) pueden ser mezclados libremente en asignaciones y expresiones. Las conversiones se ejecutan automáticamente: los operandos de tipo más bajo se convierten en los de tipo más alto.

```
int i = 12; double x = 4;
x = x+i;        /* valor de i se convierte en double antes de la suma */
x = i/5;        /* primero hace una división entera i/5=2, 2 se convierte a tipo double: 2.0
                y se asigna a x */
x = 4.0;
x = x/5;        /* convierte 5 a tipo double, hace una división real: 0.8 y se asigna a x */
```

Reglas

Si cualquier operando es de tipo `char`, `short` o enumerado se convierte en tipo `int` y si los operandos tienen diferentes tipos, la siguiente lista determina a qué tipo se convertirá (esta operación se llama *promoción integral*).

```
int
unsigned int long
unsigned long float
double
```

El tipo que viene primero en la lista se convierte en el que viene después. Por ejemplo, si los tipos operandos son `int` y `long`, el operando `int` se convierte en `long`.

```
char c = 65;    /* 65 se convierte en tipo char permitido */ char c = 10000;    /* permiti-
do, pero resultado impredecible */
```

Conversión explícita

C fuerza la *conversión explícita* de tipos mediante el operador de *molde* (*cast*). El operador de *molde* tiene el formato:

```
(tiponombre)valor     /* convierte valor a tiponombre */ (float)i; /* convierte i a float */
(int)3.4;             /* convierte 3.4 a entero, 3 */
(int*) malloc(2*16);  /* convierte el valor devuelto por malloc: void* a int*.
```

Es una conversión de apuntadores (punteros). `*/` El operador *molde* (*tipo*, *cast*) tiene la misma prioridad que otros operadores unitarios como – y `! precios = (int)19.99 + (int)11.99;`

3.11. Entrada y salida de información

Los cálculos que realizan las computadoras, para ser útiles requieren la *entrada* de los datos necesarios para ejecutar las operaciones que posteriormente se convertirán en resultados, es decir, *salida*.

Las operaciones de entrada permiten leer determinados valores y asignarlos a ciertas variables. Esta entrada se conoce como operación de **lectura** (*read*). Los datos de entrada se introducen al procesador mediante dispositivos de entrada (teclado, tarjetas perforadas, unidades de disco, etc.). La salida puede aparecer en un dispositivo de salida (pantalla, impresora, etc.). La operación de salida se denomina **escritura** (*write*).

En la escritura de algoritmos las acciones de lectura y escritura se representan por los formatos siguientes:

```
leer (lista de variables de entrada)
escribir (lista de variables de salida)
```

Así, por ejemplo:

```
leer (A, B, C)
```

representa la lectura de tres valores de entrada que se asignan a las variables `A`, `B` y `C`.

```
escribir ('hola Vargas')
```

visualiza en la pantalla —o escribe en el dispositivo de salida— el mensaje `'hola Vargas'`.

Nota 1

Si se utilizaran las palabras reservadas en inglés, como suele ocurrir en los lenguajes de programación, se deberá sustituir:

```
leer      escribir
```

por

```
read      write o bien print
```

Cristian Pilapanta 1095211-2202100102

Nota 2

Si no se especifica el tipo de dispositivo del cual se leen o escriben datos, los dispositivos predeterminados de E/S son el teclado y la pantalla.

3.12. Escritura de algoritmos/programas

La escritura de un algoritmo mediante una herramienta de programación debe ser lo más clara posible y estructurada, de modo que su lectura facilite considerablemente el entendimiento del algoritmo y su posterior codificación en un lenguaje de programación.

Los algoritmos deben ser escritos en lenguajes similares a los programas. En nuestro libro utilizaremos esencialmente el lenguaje algorítmico, basado en pseudocódigo, y la estructura del algoritmo requerirá la lógica de los programas escritos en el lenguaje de programación estructurado, por ejemplo, Pascal.

Un algoritmo constará de dos componentes: *una cabecera de programa* y *un bloque algoritmo*. La *cabecera de programa* es una acción simple que comienza con la palabra **algoritmo**. Esta palabra estará seguida por el nombre asignado al programa completo. El *bloque algoritmo* es el resto del programa y consta de dos componentes o secciones: *las acciones de declaración* y *las acciones ejecutables*.

Las *declaraciones* definen o declaran las variables y constantes que tengan nombres. Las *acciones ejecutables* son las acciones que posteriormente deberá realizar la computación cuando el algoritmo convertido en programa se ejecute.

```
algoritmo
cabecera del programa
sección de declaración
sección de acciones
```

Cristian Pilapanta 1095211-2202100104

3.12.1. Cabecera del programa o algoritmo

Todos los algoritmos y programas deben comenzar con una cabecera en la que se exprese el identificador o nombre correspondiente con la palabra reservada que señale el lenguaje. En los lenguajes de programación, la palabra reservada suele ser **program**. En Algorítmica se denomina **algoritmo**.

```
algoritmo DEMO1
```

3.12.2. Declaración de variables

En esta sección se declaran o describen todas las variables utilizadas en el algoritmo, listándose sus nombres y especificando sus tipos. Esta sección comienza con la palabra reservada **var** (abreviatura de *variable*) y tiene el siguiente formato:

```
var
   tipo-1 : lista de variables-1
   tipo-2 : lista de variables-2
      .
      .
   tipo-n : lista de variables-n
```

donde cada *lista de variables* es una variable simple o una lista de variables separadas por comas y cada *tipo* es uno de los tipos de datos básicos (**entero**, **real**, **char** o **boolean**). Por ejemplo, la sección de declaración de variables:

```
var
   entera :  Número_Empleado
   real   :  Horas
   real   :  Impuesto
   real   :  Salario
```

o de modo equivalente:

```
var
   entera :    Número_Empleado
   real   :    Horas, Impuesto, Salario
```

declara que sólo las tres variables `Hora`, `Impuesto` y `Salario` son de tipo real.

Es una buena práctica de programación utilizar nombres de variables significativos que sugieran lo que ellas representan, ya que eso hará más fácil y legible el programa.

También es buena práctica incluir breves comentarios que indiquen cómo se utiliza la variable.

```
var
   entera :  Número_Empleado // número de empleado
   real   :  Horas,          // horas trabajadas
             Impuesto,       // impuesto a pagar
             Salario         // cantidad ganada
```

Cristian Pilapanta 1096211-2202100105

3.12.3. Declaración de constantes numéricas

En esta sección se declaran todas las constantes que tengan nombre. Su formato es

```
const
   pi      = 3.141592
   tamaño  = 43
   horas   = 6.50
```

1096211220021100105

Los valores de estas constantes ya no pueden variar en el transcurso del algoritmo.

3.12.4. Declaración de constantes y variables carácter

Las constantes de carácter simple y cadenas de caracteres pueden ser declaradas en la sección del programa `const`, al igual que las constantes numéricas.

```
const
   estrella = '*'
   frase    = '12 de octubre'
   mensaje  = 'Hola mi nene'
```

Las variables de caracteres se declaran de dos modos:

1. Almacenar un solo carácter.

   ```
   var carácter : nombre, inicial, nota, letra
   ```

   Se declaran `nombre`, `inicial`, `nota` y `letra`, que almacenarán sólo un carácter.

2. Almacenar múltiples caracteres (*cadenas*). El almacenamiento de caracteres múltiples dependerá del lenguaje de programación. Así, en los lenguajes:

   VB 6.0/VB .NET (VB, Visual Basic)

   ```
   Dim var1 As String
   Var1 = "Pepe Luis García Rodríguez"
   ```

Pascal formato tipo **array** o **arreglo**.

Existen algunas versiones de Pascal, como es el caso de Turbo Pascal, que tienen implementados un tipo de datos denominado *string* (cadena) que permite declarar variables de carácter o de cadena que almacenan palabras compuestas de diferentes caracteres.

```
var nombre : string[20];    en Turbo Pascal
var cadena : nombre[20];    en pseudocódigo
```

3.12.5. Comentarios

La documentación de un programa es el conjunto de información interna, externa al programa, que facilitará su posterior mantenimiento y puesta a punto. La documentación puede ser *interna* y *externa*.

La *documentación externa* es aquella que se realiza externamente al programa y con fines de mantenimiento y actualización; es muy importante en las fases posteriores a la puesta en marcha inicial de un programa. La *documentación interna* es la que se acompaña en el código o programa fuente y se realiza con base en comentarios significativos. Estos comentarios se representan con diferentes notaciones, según el tipo de lenguaje de programación.

Visual Basic 6 / VB .NET

1. Los comentarios utilizan un apóstrofo simple y el compilador ignora todo lo que viene después de ese carácter:

```
'Éste es un comentario de una sola línea
Dim Mes As String 'comentario después de una línea de código
.................
```

2. También se admite, por guardar compatibilidad con versiones antiguas de Basic y Visual Basic, la palabra reservada `Rem`:

```
Rem esto es un comentario
```

Cristian Palpanta 1095211-2202100106

C/C++ y C#

Existen dos formatos de comentarios en los lenguajes C y C++:

1. Comentarios de una línea (comienzan con el carácter //)

```
// Programa 5.0 realizado por el maestro Mackoy
// en Carchelejo (Jaén)en las fiestas de agosto
// de moros y cristianos
```

2. Comentarios multilínea (comienzan con los caracteres /* y terminan con los caracteres */, todo lo encerrado entre ambos juegos de caracteres son comentarios)

```
/* El maestro Mackoy estudió el bachillerato en el mismo instituto donde dio clase Don An-
tonio Machado, el poeta */
```

Java

1. Comentarios de una línea

```
// comentarios sobre la Ley de Protección de Datos
```

2. Comentarios multilínea

```
/* El pueblo del maestro Mackoy está en Sierra Mágina, y produce uno de los mejores aceites
de oliva del mundo mundial      */
```

3. Documentación de clases

```
/**
   Documentación de la clase
*/
```

Pascal

Los comentarios se encierran entre los símbolos

```
(*     *)
```

o bien

```
{       }
(* autor J. R. Mackoy *)
{subrutina ordenación}
```

Nota

A lo largo del libro utilizaremos preferentemente para representar nuestros comentarios los símbolos // y /*. Sin embargo, algunos autores de algoritmos, a fin de independizar la simbología del lenguaje, suelen representar los comentarios con corchetes ([ ]).

Cristian Pilapanta 1095211-2202100107

3.12.6. Estilo de escritura de algoritmos/programas

El método que seguiremos normalmente a lo largo del libro para escribir algoritmos será el descrito al comienzo de la sección 3.11.

```
algoritmo identificador  // cabecera
// sección de declaraciones
var tipo de datos : lista de identificadores
const lista de identificadores = valor
inicio
   <sentencia S1>
   <sentencia S2>       // cuerpo del algoritmo
      .
      .
      .
   <sentencia Sn>
fin
```

Notas

1. En ocasiones, la declaración de constantes y variables las omitiremos o se describirán en una tabla de variables que hace sus mismas funciones.
2. Las cadenas de caracteres se encerrarán entre comillas simples.
3. Utilizar siempre sangrías en los bucles o en aquellas instrucciones que proporcionen legibilidad al programa, como `inicio` y `fin`.

Modelo propuesto de algoritmo

```
algoritmo raíces
// resuelve una ecuación de 2.º grado
var
   real : a, b, c
inicio
   leer(a, b, c)
   d ←b ^ 2 - 4 * a * c
   si d < 0 entonces
      escribir('raíces complejas')
   si_no
      si d = 0 entonces
         escribir (-b / (2 * a)
      si_no
         escribir ((-b - raiz2(d)) / (2 * a)
         escribir ((-b + raíz2(d)) / (2 * a)
      fin_si
   fin_si
fin
```

Cristian Palpanta 1095211-2202100108

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

3.1. *Diseñar un algoritmo para cambiar una rueda de un coche.*

Solución

```
algoritmo pinchazo
inicio
   si gato del coche está averiado
      entonces llamar a la estación de servicio
      si_no levantar el coche con el gato
         repetir
           aflojar y sacar los tornillos de las ruedas
         hasta_que todos los tornillos estén flojos y quitados
         quitar la rueda
         poner la rueda de repuesto
         repetir
            poner los tornillos y apretarlos
         hasta_que estén puestos todos los tornillos
         bajar el gato
   fin_si
fin
```

3.2. *Encontrar el valor de la variable* VALOR *después de la ejecución de las siguientes operaciones:*

```
(A)   VALOR ←4.0 * 5
(B)   X ←3.0
      Y ←2.0
      VALOR ←X ^ Y - Y
(C)   VALOR ←5
      X ←3
      VALOR ←VALOR * X
```

Solución

```
(A)  VALOR = 20.0
(B)  X = 3.0
     Y = 2.0
     VALOR = 3 * 2 - 2 = 9 - 2 = 7     VALOR = 7
(C)  VALOR = 5
     X = 3
     VALOR = VALOR * X = 5 * 3 = 15    VALOR = 15
```

3.3. *Deducir el resultado que se produce con las siguientes instrucciones:*

```
var Entero : X, Y
X ← 1
Y ← 5
escribir (X, Y)
```

Solución

X e Y toman los valores 1 y 5. La instrucción de salida (`escribir`) presentará en el dispositivo de salida 1 y 5, con los formatos específicos del lenguaje de programación; por ejemplo,

```
1    5
```

3.4. *Deducir el valor de las expresiones siguientes:*

```
X ← A + B + C
X ← A + B * C
X ← A + B / C
X ← A + B \ C
X ← A + B mod C
X ← (A + B) \ C
X ← A + (B / C)
Siendo A = 5 B = 25 C = 10
```

Solución

Expresión	X
A + B + C = 5 + 25 + 10	40
A + B * C = 5 + 25 * 10	225
A + B / C = 5 + 25 / 10	7.5
A + B \ C = 5 + 25 \ 10 = 5 + 2	7
A + B mod C = 5 + 25 mod 10 = 5 + 5	10
(A + B) / C = (5 + 25) / 10 = 30 / 10	3
A + (B / C) = 5 + (25 / 10) = 5 + 2.5	7.5

3.5. *Escribir las siguientes expresiones en forma de expresiones algorítmicas:*

a) $\frac{M}{N} + P$

b) $M + \frac{N}{P - Q}$

c) $\frac{seno(x) + cos(x)}{tan(x)}$

d) $\frac{m + n}{p - q}$

e) $\frac{m + \frac{n}{p}}{q - \frac{r}{5}}$

f) $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$

Cristian Pliqueria 1095211-2202100109

Solución

a) `M / N + P`
b) `M + N / (P - Q)`
c) `(SEN(X) + COS(X)) / TAN (X)`
d) `(M + N) / (P - Q)`
e) `(M + N / P) / (Q - R / 5)`
f) `(-B + raiz2 (B ^ 2 - 4 * A * C)) / (2 * A)`

3.6. *Calcúlese el valor de las siguientes expresiones:*

a) $8 + 7 * 3 + 4 * 6$
b) $-2 \wedge 3$
c) $(33 + 3 * 4) / 5$
d) $2 \wedge 2 * 3$
e) $3 + 2 * (18 - 4 \wedge 2)$
f) $16 * 6 - 3 * 2$

Solución

a) `8 + 7 * 3 + 4 * 6`
`8 + 21 + 24`
`29 + 24`
`53`

b) `-2 ^ 3`
`-8`

c) `(33 + 3 * 4) / 5`
`33 + 12 / 5`
`45 / 5`
`9`

d) `2 ^ 2 * 3`
`4 * 3`
`12`

e) `16 * 6 - 3 * 2`
`96 - 6`
`90`

3.7. *Se tienen tres variables A, B y C. Escribir las instrucciones necesarias para intercambiar entre sí sus valores del modo siguiente:*

- *B toma el valor de A*
- *C toma el valor de B*
- *A toma el valor de C*

Nota: Sólo se debe utilizar una variable auxiliar.

Solución

Utilizaremos una variable auxiliar `AUX`.

Cristian Pilapanta 1006211-2202100109
1006211220210010

Las instrucciones que resuelven el problema de intercambio son:

```
AUX ← A
  A ← C
  C ← B
  B ← AUX
```

Comprobémoslo con los valores de A, B y C: 5, 10 y 15.

	Instrucción	A	B	C	AUX	Observaciones
(1)	A ← 5	5	--	--	--	
(2)	B ← 10	--	10	--	--	
(3)	C ← 15	--	--	15	--	
	AUX ← A	5	10	15	5	
	A ← C	15	10	15	5	
	C ← B	15	10	10	5	
	B ← AUX	15	5	10	5	

Observe que al igual que en el ejercicio de intercambio de valores entre dos variables, la variable AUX no modifica su valor.

3.8. *Cómo se intercambian los valores de dos variables, A y B.*

Solución

Como hemos visto en este ejercicio, se pueden intercambiar los valores de una variable mediante las instrucciones:

```
A ← B
B ← A
```

El procedimiento para conseguir intercambiar los valores de dos variables entre sí debe recurrir a una variable AUX y a las instrucciones de asignación siguientes:

```
AUX ← A
  A ← B
  B ← AUX
```

Veámoslo con un ejemplo:

```
a ← 10
B ← 5
```

Instrucción	A	B	AUX	Observaciones
A ← 10	10	--	--	
B ← 5	10	5	--	
AUX ← A	10	5	10	La variable AUX toma el valor de A
A ← B	5	5	10	A toma el valor de B, 5
B ← AUX	5	10	10	B toma el valor inicial de A, 10

Ahora A = 5 y B = 10.

3.9. *Deducir el valor que toma la variable tras la ejecución de las instrucciones:*

```
A ← 4
B ← A
B ← A + 3
```

Cristian Pilapanta 1006211-2202100110

Solución

Mediante una tabla se da un método eficaz para obtener los sucesivos valores:

```
                          A     B
(1)   A ← A               4     --
(2)   B ← A               4     4
(3)   B ← A + 3           4     7
```

Después de la instrucción (1) la variable A contiene el valor 4.

La variable B no ha tomado todavía ningún valor y se representa esa situación con un guion.

La instrucción (2) asigna el valor actual de A (4) a la variable B. La instrucción (3) efectúa el cálculo de la expresión A + 3, lo que produce un resultado de 7 (4 + 3) y este valor se asigna a la variable B, cuyo último valor (4) se destruye.

Por consiguiente, los valores finales que tienen las variables A y B son:

```
A = 4      B = 7
```

3.10. *¿Qué se obtiene en las variables A y B, después de la ejecución de las siguientes instrucciones?*

```
A ← 5
B ← A + 6
A ← A + 1
B ← A - 5
```

Solución

Siguiendo las directrices del ejercicio anterior:

	Instrucción	A	B	Observaciones
(1)	A ← 5	5	—	B no toma ningún valor
(2)	B ← A + 6	5	11	Se evalúa A + 6 (5 + 6) y se asigna a B
(3)	A ← A + 1	6	11	Se evalúa A + 1 (5 + 1) y se asigna a A, borrándose el valor que tenía (5) y tomando el nuevo valor (6)
(4)	B ← A - 5	6	1	Se evalúa A - 5 (6 - 1) y se asigna a B

Los valores últimos de A y B son: A = 6, B = 1.

3.11. *¿Qué se obtiene en las variables A, B y C después de ejecutar las instrucciones siguientes?*

```
A ← 3
B ← 20
C ← A + B
B ← A + B
A ← B - C
```

Solución

	Instrucción	A	B	C	Observaciones
(1)	A ← 3	3	--	--	B y C no toman ningún valor
(2)	B ← 20	3	20	--	C sigue sin valor
(3)	C ← A + B	3	20	23	Se evalúa A + B (20 + 3) y se asigna a C
(4)	B ← A + B	3	23	23	Se evalúa A + B (20 + 3) y se asigna a B; destruye el valor antiguo (20)
(5)	A ← B - C	0	23	23	Se evalúa B - C (23 - 23) y se asigna a A

Cristian Pilapanta 1095211-2202100112

Los valores finales de las variables son:

A = 0 B = 23 C = 23

3.12. *¿Qué se obtiene en A y B tras la ejecución de lo siguiente?*

```
A ← 10
B ← 5
B ← B
B ← A
```

Solución

	Instrucción	A	B	Observaciones
(1)	A ← 10	10	--	B no toma valor
(2)	B ← 5	10	5	B recibe el valor inicial 5
(3)	A ← B	5	5	A toma el valor de B (5)
(4)	B ← A	5	5	B toma el valor actual de A (5)

Los valores finales de A y B son 5. En este caso se podría decir que la instrucción (4) B ← A es *redundante* respecto a las anteriores, ya que su ejecución no afecta al valor de las variables.

Cristian Palpanta 1095211-2202100113

3.13. *Determinar el mayor de tres números enteros.*

Solución

Los pasos a seguir son:

1. Comparar el primero y el segundo entero, deduciendo cuál es el mayor.
2. Comparar el mayor anterior con el tercero y deducir cuál es el mayor. Éste será el resultado.

Los pasos anteriores se pueden descomponer en otros pasos más simples en lo que se denomina *refinamiento del algoritmo*:

1. Obtener el primer número (entrada), denominarlo NUM1.
2. Obtener el segundo número (entrada), denominarlo NUM2.
3. Comparar NUM1 con NUM2 y seleccionar el mayor; si los dos enteros son iguales, seleccionar NUM1. Llamar a este número MAYOR.
4. Obtener el tercer número (entrada) y denominarlo NUM3.
5. Comparar MAYOR con NUM3 y seleccionar el mayor; si los dos enteros son iguales, seleccionar el MAYOR. Denominar a este número MAYOR.
6. Presentar el valor de MAYOR (salida).
7. Fin.

3.14. *Determinar la cantidad total a pagar por una llamada telefónica, teniendo en cuenta lo siguiente:*

- *Toda llamada que dure menos de tres minutos (cinco pasos) tiene un costo de 10 céntimos.*
- *Cada minuto adicional a partir de los tres primeros es un paso de contador y cuesta 5 céntimos.*

Solución

Análisis

El algoritmo de resolución del problema entraña los siguientes pasos:

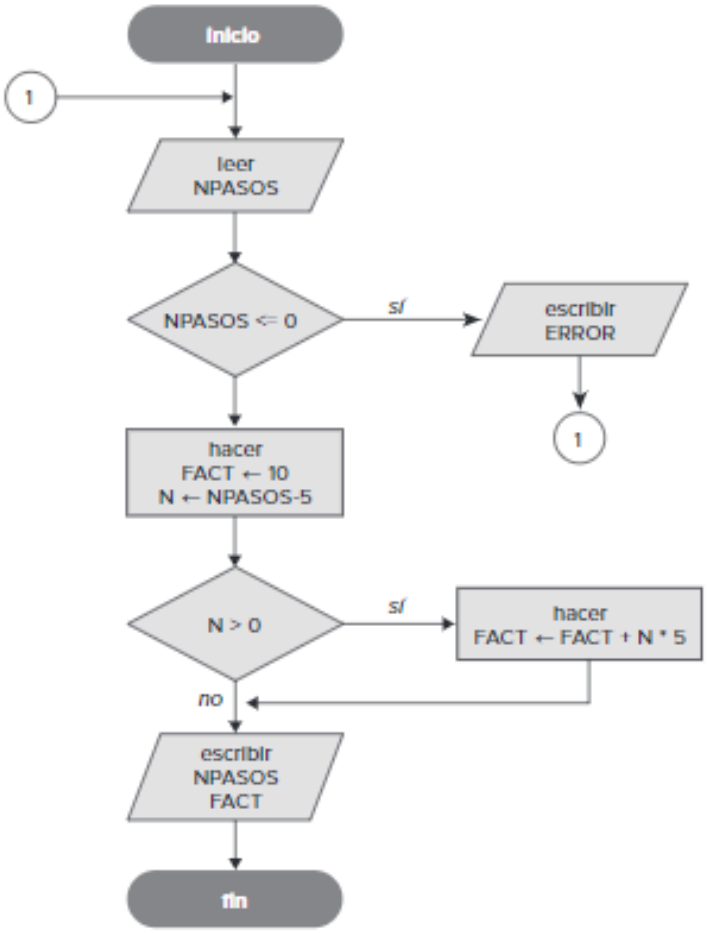
1. Inicio.
2. Leer el número se pasos (npasos) hablados por teléfono.
3. Comprobar que el número de pasos es mayor que cero, ya que realmente se ha realizado la llamada si el número de pasos es distinto de cero (positivo). Si el número de pasos es menor que cero, se producirá un error.

4. Calcular el precio de la conferencia de acuerdo con los siguientes conceptos:
   - Si el número de pasos es menor que 5, el precio es de 10 centavos,
   - Si el número de pasos es mayor que 5, es preciso calcular los pasos que exceden de 5, ya que éstos importan 5 centavos cada uno; al producto de los pasos sobrantes por 5 centavos se le suman 10 pesetas y se obtendrá el precio total.

Variables

```
NPASOS  Número de pasos de la llamada
N       Número de pasos que exceden a 5
FACT    Importe o precio de la llamada
```

Diagrama de flujo



Cristian Pilapanta 1095211-2202100114

3.15. *Calcular la suma de los cincuenta primeros números enteros.*

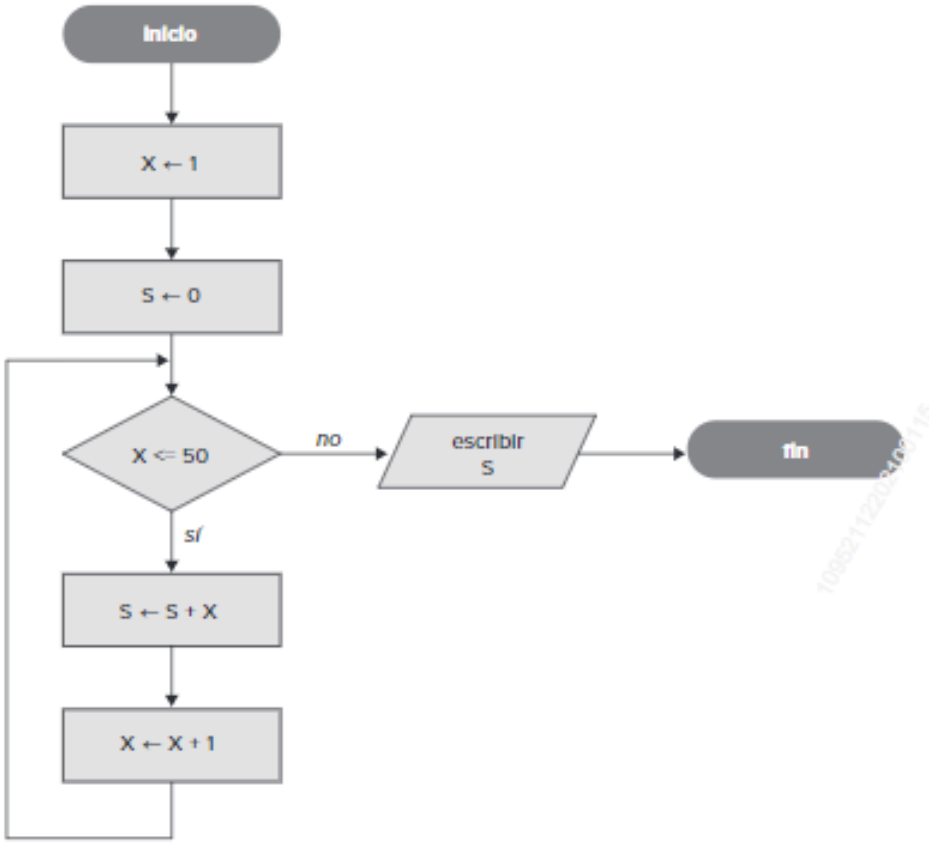
Solución

Análisis

El algoritmo expresado en lenguaje natural o en secuencia de pasos es el siguiente:

1. Inicio.
2. Hacer el primer número 1 igual a una variable X que actuará de contador de 1 a 50 y S igual a 0.
3. Hacer S = S+X para realizar las sumas parciales.
4. Hacer X = X+1 para generar los números enteros.
5. Repetir los pasos 3 y 4 hasta que X = 50, en cuyo caso se debe visualizar la suma.
6. Fin.

Diagrama de flujo



3.16. *Escribir un algoritmo que calcule el producto de los n primeros números naturales.*

Solución

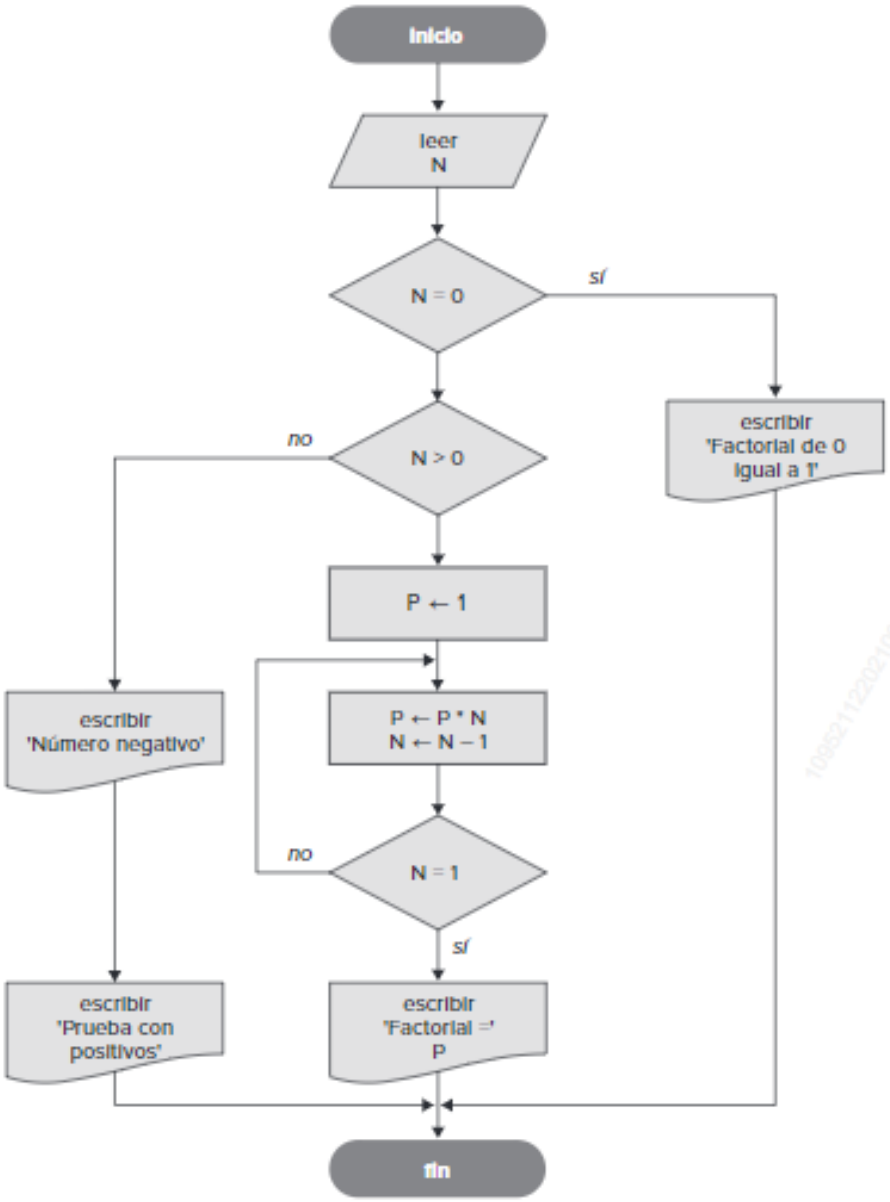
Análisis

El problema puede calcular el producto N * (N - 1 * (n - 2) * ... * 3 * 2 * 1, que en términos matemáticos se le conoce con el nombre de FACTORIAL de N. El algoritmo que resuelve el problema es el siguiente:

1. Leer N.
2. En caso de que N = 0, visualizar «Factorial de 0 igual 1».
3. Comprobar que N > 0 (los números negativos no se consideran).
4. Hacer la variable P que va a contener el productor igual a 1.
5. Realizar el producto P = P * N.
   Disminuir en una unidad sucesivamente hasta llegar a N = 1, y de modo simultáneo los productos P * N.
6. Visualizar P.
7. Fin.

Cristian Pliqueria 1096211-220210115

Diagrama de flujo



Cristian Pilapanta 1006211-2202100115

Pseudocódigo

```
algoritmo Factorial
var
      entero : N
      real   : P
inicio
      leer(N)
      si N = 0 entonces
         escribir('Factorial de 0 igual a 1')
```

```
        si_no
           si N > 0 entonces
               P ← 1
           (1) P ← P * N
               N ← N - 1
           si N = 1 entonces
              escribir('Factorial =', P)
           si_no
              ir_a (1)
           fin_si
        si_no
           escribir('Número negativo')
           escribir('Pruebe con positivos')
        fin_si
     fin_si
  fin
```

3.17. *Diseñar un algoritmo para resolver una ecuación de segundo grado* $Ax^2 + Bx + C = 0$.

Solución

Análisis

La ecuación de segundo grado es $Ax^2 + Bx + C = 0$ y las soluciones o raíces de la ecuación son:

$$X1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \qquad X2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

Para que la ecuación de segundo grado tenga solución, es preciso que el discriminante sea mayor o igual que 0. El discriminante de una ecuación de segundo grado es:

```
D = B ^ 2 - 4AC
```

Por consiguiente, si:

```
D = 0     X1 = -B / 2A          X2 = -B / 2A
D < 0     X1 y X2
```

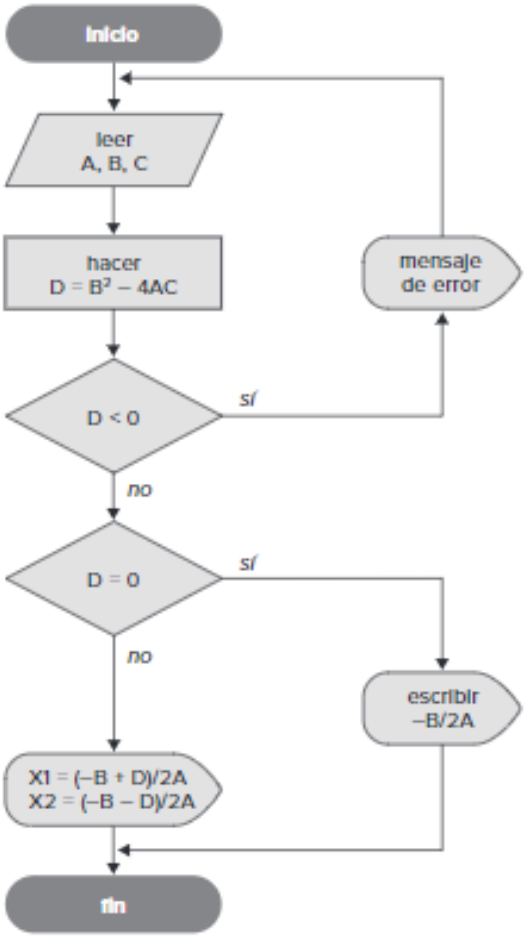
no tienen solución real.

En consecuencia, el algoritmo que resolverá el problema es el siguiente:

1. Inicio.
2. Introducir los coeficientes A, B y C.
3. Cálculo del discriminante D = B ^ 2 - 4AC
4. Comprobar el valor de D.
   - si D es menor que 0, visualizar un mensaje de error,
   - si D es igual que 0, se obtienen dos raíces iguales X1 = X2 = -B / 2A.
   - si D es mayor que 0, se calculan las dos raíces X1 y X2.
5. Fin del algoritmo.

Cristian Pilapanta 1096211-2202100116

Diagrama de flujo



Cristian Pilapanta 1096211-2202100117

3.18. *Escribir un algoritmo que acepte tres números enteros e imprima el mayor de ellos.*

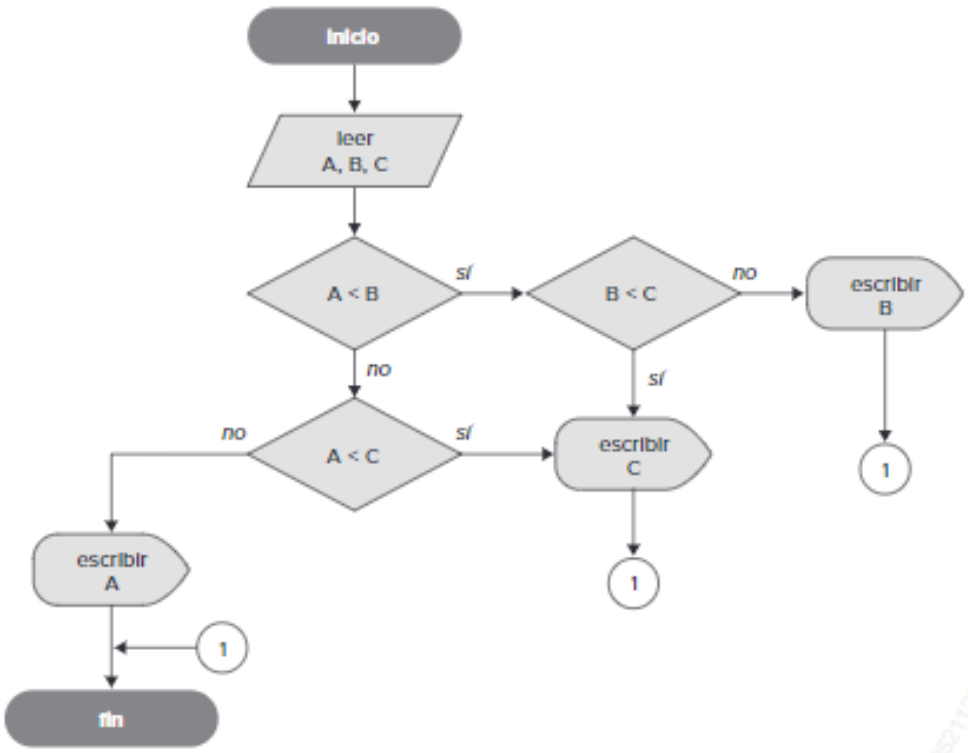
Solución

Análisis

El diseño del algoritmo requiere de una serie de comparaciones sucesivas. Las operaciones sucesivas son las siguientes:

```
1. Inicio.
2. Introducir los tres números A, B, C.
3. Comparar A y B:
   • si A es menor que B:
     - comparar B y C:
       • si B es mayor que C, el mayor es B,
       • si B es menor que C, el mayor es C.
   • si A es mayor que B:
     - comparar A y C:
       • si A es menor que C, el mayor es C,
       • si A es mayor que C, el mayor es A.
```

Diagrama de flujo



Cristian Pilapanta 1095211-2202100123

10952112202100123

Conceptos clave

- Algoritmo
- Asignación
- Caracteres especiales
- Constantes
- Datos
- Declaraciones
- Escritura de resultados
- Expresiones
- Función interna
- Identificador
- Instrucción
- Lectura de datos
- Operaciones primitivas
- Operadores
- Palabras reservadas
- Programa
- Pseudocódigo
- Tipos de datos
- Variables

Resumen

Un programa es un conjunto de instrucciones que se proporciona a una computadora para realizar una tarea determinada. El proceso de programación requiere las siguientes fases o etapas fundamentales: *definición y análisis del problema, diseño del algoritmo, codificación del programa, depuración y verificación, documentación y mantenimiento.*

En la práctica un programa es una caja negra —un algoritmo de resolución del problema— que tiene una entrada de datos y una salida de resultados. La entrada de datos se realiza a través del teclado, ratón, escáner, discos... y la salida se representa en impresora, pantalla, etcétera.

Existen diferentes tipos de instrucciones básicas: *inicio, fin, asignación, lectura, escritura y bifurcación.*

Los elementos básicos constitutivos de un programa son: *palabras reservadas, identificadores, caracteres especiales, constantes, variables, expresiones, instruccio-*

nes a los cuales se unen para tareas de ejecución de operaciones otros elementos primitivos de un programa, tales como: *bucles, contadores, acumuladores, interruptores y estructuras.* Todos estos elementos manipulan datos o información de diferentes tipos como *numéricos, lógicos o carácter.* Los valores de estos datos se almacenan para su tratamiento en constantes y variables. Las combinaciones de constantes, variables, símbolos de operaciones, nombres de funciones, etc., constituyen las expresiones que a su vez se clasifican en función del tipo de objetos que manipulan en: *aritméticas, relacionales, lógicas y carácter.*

Otro concepto importante a considerar en la iniciación a la programación es el concepto y tipos de operadores que sirven para la resolución de expresiones y constituyen elementos clave en las sentencias de flujo de control que se estudiarán en los capítulos posteriores.

La operación de asignación es un sistema de almacenamiento de valores en una variable. Existen diferentes tipos de asignaciones en función de los tipos de datos que se desean almacenar. La conversión de tipos en operaciones de asignaciones es una tarea importante y su comprensión es vital para evitar errores en el proceso de depuración de un programa.

La última característica importante a considerar en el capítulo es la escritura de algoritmos y programas, para lo que se necesitan unas reglas claras y precisas que faciliten su legibilidad y su posterior codificación en un lenguaje de programación.

Cristian Palpanta 1006211-2202150123

EJERCICIOS

3.1. Diseñar los algoritmos que resuelvan los siguientes problemas:

a) Ir al cine.
b) Comprar una entrada para los toros.
c) Colocar la mesa para comer.
d) Cocer un huevo.
e) Hacer una taza de té.
f) Fregar los platos del almuerzo.
g) Buscar el número de teléfono de un alumno.
h) Reparar un pinchazo de una bicicleta.
i) Pagar una multa de tráfico.
j) Cambiar un neumático pinchado (se dispone de herramientas y gato).
k) Hacer palomitas de maíz en una olla puesta al fuego con aceite, sal y maíz.
l) Cambiar el cristal roto de una ventana.
m) Hacer una llamada telefónica. Considerar los casos: a) manual, con operadora; b) automático; c) cobro revertido.
n) Quitar una bombilla quemada de un techo.
o) Encontrar la media de una lista indeterminada de números positivos terminada con un número negativo.

3.2. ¿Cuáles de los siguientes identificadores no son válidos?

a) `XRayo`
b) `X_Rayo`
c) `R2D2`
d) `X`
e) `45`
f) `N14`
g) `ZZZZ`
h) `3µ`

3.3. ¿Cuáles de las siguientes constantes no son válidas?

a) `234`
b) `-8.975`
c) `12E - 5`
d) `0`
e) `32,767`
f) `1/2`
g) `3.6E + 7`
h) `7E12`
i) `3.5 × 10`
j) `0.456`
k) `0.000001`
l) `224E1`

3.4. Evaluar la siguiente expresión para `A = 2` y `B = 5`:

```
3 * A - 4 * B / A ^ 2
```

3.5. Evaluar la siguiente expresión:

```
4 / 2 * 3 / 6 + 6 / 2 / 1 / 5 ^ 2 / 4 * 2
```

3.6. Escribir las siguientes expresiones algebraicas como expresiones algorítmicas:

a) $\sqrt{b^2 - 4ac}$
b) $\frac{x^2 + y^2}{z^2}$
c) $\frac{3x + 2y}{2z}$
d) $\frac{a + b}{c - d}$
e) $4x^2 - 2x + 7$
f) $\frac{x + y}{x} - \frac{3x}{5}$
g) $\frac{a}{bc}$
h) xyz
i) $\frac{y_2 - y_1}{x_2 - x_1}$
j) $2\pi r$

k) $\frac{4}{3}\pi r^3$ l) $(x_2 - x_1)^2 + (y_2 - y_1)^2$

3.7. Escribir las siguientes expresiones algorítmicas como expresiones algebraicas:

a) `b ^ 2 - 4 * a * c`
b) `3 * X ^ 4 - 5 * X ^ 3 + X 12 - 17`
c) `(b + d) / (c + 4)`
d) `(x ^ 2 + y ^ 2) ^ (1 / 2)`

3.8. Si el valor de `A` es 4, el valor de `B` es 5 y el valor de `C` es 1, evaluar las siguientes expresiones:

a) `B * A - B ^ 2 / 4 * C`
b) `(A * B) / 3 ^ 2`
c) `(((B + C) / 2 * A + 10) * 3 * B) - 6`

3.9. Si el valor de `A` es 2, `B` es 3 y `C` es 2, evaluar la siguiente expresión:

```
A ^ B ^ C
```

3.10. Obtener el valor de cada una de las siguientes expresiones aritméticas:

a) `7 div 2`
b) `7 mod 2`
c) `12 div 3`
d) `12 mod 3`
e) `0 mod 5`
f) `15 mod 5`
g) `7 * 10 - 50 mod 3 * 4 + 9`
h) `(7 * (10 - 5) mod 3) * 4 + 9`

Nota: Considere la prioridad de Pascal: más alta: `*`, `/`, `div`, `mod`; más baja: +, -.

3.11. Encontrar el valor de cada una de las siguientes expresiones, o decir si no es una expresión válida:

a) `9 - 5 - 3`
b) `2 div 3 + 3 / 5`
c) `9 div 2 / 5`
d) `7 mod 5 mod 3`
e) `7 mod (5 mod 3)`
f) `(7 mod 5) mod 3`
g) `(7 mod 5 mod 3)`
h) `((12 + 3) div 2) / (8 - (5 + 1))`
i) `12 / 2 * 3`
j) `raiz2 (cuadrado(4)`
k) `cuadrado (raiz2(4))`
l) `trunc(815) + redondeo(815)`

Considérese la prioridad del ejercicio 3.10.

3.12. Se desea calcular independiente la suma de los números pares e impares comprendidos entre 1 y 200.

3.13. Leer una serie de números distintos de cero (el último número de la serie es –99) y obtener el número mayor. Como resultado se debe visualizar el número mayor y un mensaje de indicación de número negativo, caso de que se haya leído un número negativo.

3.14. Calcular y visualizar la suma y el producto de los números pares comprendidos entre 20 y 400, ambos inclusive.

3.15. Leer 500 números enteros y obtener cuántos son positivos.

3.16. Escribir el algoritmo que permita emitir la factura correspondiente a la compra de un artículo determinado del que se adquieren una o varias unidades. El IVA a aplicar es del 15% y si el precio bruto (precio venta más IVA) es mayor que 1 000 euros, se debe realizar un descuento de 5%.

3.17. Calcular la suma de los cuadrados de los 100 primeros números naturales.

3.18. Sumar los números pares del 2 al 100 e imprimir su valor.

3.19. Sumar 10 números introducidos por teclado.

3.20. Calcular la media de 50 números e imprimir su resultado.

3.21. Calcular los *N* primeros múltiplos de 4 (4 inclusive), donde *N* es un valor introducido por teclado.

3.22. Diseñar un diagrama que permita realizar un contador e imprimir los 100 primeros números enteros.

3.23. Dados 10 números enteros, visualizar la suma de los números pares de la lista, cuántos números pares existen y cuál es la media aritmética de los números impares.

3.24. Calcular la nota media de los alumnos de una clase considerando n-número de alumnos y c-número de notas de cada alumno.

3.25. Escribir la suma de los 10 primeros números pares.

3.26. Escribir un algoritmo que lea los datos de entrada de un archivo que sólo contiene números y sume los números positivos.

3.27. Desarrollar un algoritmo que determine en un conjunto de 100 números naturales:

- ¿Cuántos son menores de 15?
- ¿Cuántos son mayores de 50?
- ¿Cuántos están comprendidos entre 25 y 45?

Cristian Pulgarín 1065211-2202100123

1095211220210012

CAPÍTULO 4

Flujo de control I: estructuras selectivas

4.1. El flujo de control de un programa
4.2. Estructura secuencial
4.3. Estructuras selectivas
4.4. Alternativa simple (`si-entonces/if-then`)
4.5. Alternativa múltiple (`según_sea, caso de; switch-case`)
4.6. Estructuras de decisión anidadas (en escalera)
4.7. La sentencia `ir-a (goto)` [`no recomendable su uso, con excepciones`]
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Cristian Pilapanta 1086211-2202100125

1086211220210012S

INTRODUCCIÓN

En la actualidad, dado el tamaño considerable de las memorias centrales y las altas velocidades de los procesadores —**Intel Core** i9, i7, i5, i3...; **AMD** gamas Athlon, y Ryzen 2950X, 3700X, 3900X, etcétera—, *el estilo de escritura de los programas se vuelve una de las características más sobresalientes en las técnicas de programación. La legibilidad de los algoritmos y posteriormente de los programas* exige que su diseño sea fácil de comprender y su flujo lógico, fácil de seguir. La *programación modular* enseña la descomposición de un programa en módulos más simples de programar, y la *programación estructurada* permite la escritura de programas fáciles de leer y modificar. En *un programa estructurado* el flujo lógico se gobierna por las estructuras de control básicas:

1. *Secuenciales.*
2. *Repetitivas.*
3. *Selectivas.*

En este capítulo se introducen las estructuras selectivas que se utilizan para controlar el orden en que se ejecutan las sentencias de un programa. Las sentencias `si` (en inglés, "`if`") y sus variantes, `si-entonces`, `si-entonces-sino` y la sentencia `según-sea` (en inglés, "`switch`") se describen como parte fundamental de un programa. Las sentencias `si` anidadas y las de multibifurcación pueden ayudar a resolver importantes problemas de cálculo. Asimismo, se describe la "tristemente famosa" sentencia `ir-a` (en inglés "`goto`"), cuyo uso se debe evitar en la mayoría de las situaciones, pero cuyo significado debe ser muy bien entendido por el lector, precisamente para evitar su uso, aunque puede haber una situación específica en que no quede otro remedio que recurrir a ella.

El estudio de las estructuras de control se realiza basado en las herramientas de programación ya estudiadas: diagramas de flujo, diagramas N-S y pseudocódigos.

4.1. El flujo de control de un programa

Muchos avances han ocurrido en los fundamentos teóricos de programación desde la aparición de los lenguajes de alto nivel a finales de la década de 1950. Uno de los más importantes avances fue el reconocimiento a finales de los sesenta de que cualquier algoritmo, no importaba su complejidad, podía ser construido utilizando combinaciones de tres estructuras de control de flujo estandarizadas (*secuencial, selección, repetitiva* o *iterativa*) y una cuarta denominada *invocación* o *salto (jump)*. Las sentencias de *selección son:* `si (if)` y `según-sea (switch)`; las *sentencias de repetición o iterativas* son: `desde (for)`, `mientras (while)`, `hacer-mientras (do-while)` o `repetir-hasta que (repeat-until)`; las sentencias de salto o bifurcación incluyen `romper (break)`, `continuar (continue)`, `ir-a (goto)`, `volver (return)` y `lanzar (throw)`.

El término **flujo de control** se refiere al orden en que se ejecutan las sentencias del programa. Otros términos utilizados son *secuenciación* y *control del flujo*. A menos que se especifique expresamente, el flujo normal de control de todos los programas es **secuencial**. Este término significa que las sentencias se ejecutan en secuencia, una después de otra, en el orden en que se sitúan dentro del programa. Las estructuras de selección, repetición o iteración permiten que el flujo secuencial del programa sea modificado en un modo preciso y definido con anterioridad. Como se puede deducir fácilmente, las estructuras de selección se utilizan para seleccionar cuáles sentencias se han de ejecutar a continuación y las estructuras de repetición (repetitivas o iterativas) se utilizan para repetir un conjunto de sentencias.

Hasta este momento, todas las sentencias se ejecutaban secuencialmente en el orden en que estaban escritas en el código fuente; esto se denomina *ejecución secuencial*. Un programa basado en este tipo de ejecución, siempre ejecutará exactamente las mismas acciones; es incapaz de reaccionar en respuesta a condiciones actuales. Sin embargo, la vida real no es tan simple. Normalmente, los programas necesitan alterar o modificar el flujo de control en un programa. Así, en la solución de muchos problemas se deben tomar acciones diferentes dependiendo del valor de los datos. Ejemplos de situaciones simples son: cálculo de una superficie *sólo si* las medidas de los lados son positivas; la ejecución de una división se realiza, *sólo si* el divisor no es cero; la visualización de mensajes diferentes *depende* del valor de una nota recibida, etcétera.

Una ***bifurcación*** (*branch*, en inglés) es un segmento de programa construido con una o varias sentencias. Una *sentencia de bifurcación* se utiliza para ejecutar una sentencia de entre varias o bien bloques de ellas. La elección se realiza dependiendo de una condición dada. Las *sentencias de bifurcación* se llaman también *sentencias de selección, de alternación o alternativas.*

Cristian Pilapanta 1096211-2202100126

4.2. Estructura secuencial

Una **estructura secuencial** es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el final del proceso. La estructura secuencial tiene una entrada y una salida. Su representación gráfica se muestra en las figuras 4.1, 4.2 y 4.3.

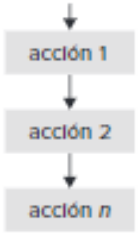


Figura 4.1. Estructura secuencial.

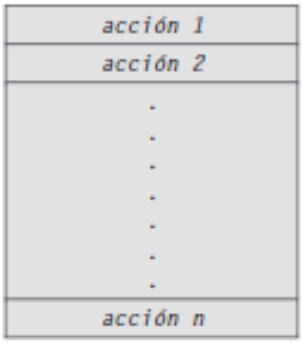


Figura 4.2. Diagrama N-S de una estructura secuencial.

```
inicio
  <acción 1>
  <acción 2>
fin
```

Figura 4.3. Pseudocódigo de una estructura secuencial.

EJEMPLO 4.1

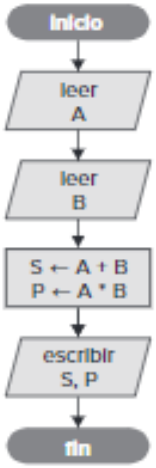
Cálculo de la suma y producto de dos números.

La suma S de dos números es S = A + B y el producto P es P = A * B. El pseudocódigo y el diagrama de flujo correspondientes se muestran a continuación:

Pseudocódigo

```
inicio
   leer(A)
   leer(B)
   S ← A + B
   P ← A * B
   escribir(S, P)
fin
```

Diagrama de flujo



Cristian P Hispania 1095211-2202100127

1095211220210012?

EJEMPLO 4.2

Se trata de calcular el salario neto de un trabajador en función del número de horas trabajadas, precio de la hora de trabajo y, considerando unos descuentos fijos, el sueldo bruto en concepto de impuestos (20 por 100).

Pseudocódigo

```
inicio
   // cálculo salario neto leer(nombre,
   horas, precio_hora) salario_bruto ←
   horas * precio_hora impuestos ← 0.20
   * salario_bruto
   salario_neto ← salario_bruto - impuestos
   escribir(nombre, salario_bruto, salario_neto)
fin
```

Diagrama de flujo

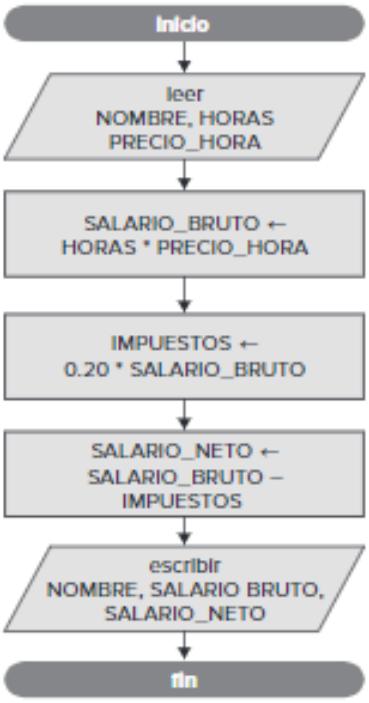


Diagrama N-S

```
leer
nombre, horas, precio
salario_bruto ← horas * precio
impuestos ← 0.20 * salario_bruto
salario_neto ← salario_bruto - impuestos
escribir
nombre, salario_bruto, salario_neto
```

Cristian Palpanta 1095211-2202100128

1095211220210012B

4.3. Estructuras selectivas

La especificación formal de algoritmos tiene realmente utilidad cuando el algoritmo requiere una descripción más complicada que una lista sencilla de instrucciones. Éste es el caso cuando existe un número de posibles alternativas resultantes de la evaluación de una determinada condición. Las estructuras selectivas se utilizan para tomar decisiones lógicas; de ahí que se denominen también *estructuras de decisión o alternativas*.

En las estructuras selectivas se evalúa una condición y en función del resultado de la misma se realiza una opción u otra. Las condiciones se especifican usando expresiones lógicas. La representación de una estructura selectiva se hace con palabras en pseudocódigo (`if`, `then`, `else` o bien en español `si`, `entonces`, `si_no`), con una figura geométrica en forma de rombo o bien con un triángulo en el interior de una caja rectangular. Las estructuras selectivas o alternativas pueden ser:

- *simples.*
- *dobles.*
- *múltiples.*

La estructura simple es **si** (**if**) con dos formatos: *Formato* Pascal, **si-entonces** (**if-then**) y *formato C*, **si** (**if**). La estructura selectiva doble es igual que la estructura simple **si**, a la cual se le añade la cláusula **si-no** (**else**). La estructura selectiva múltiple es **según_sea** (**switch** en lenguaje C, **case** en Pascal).

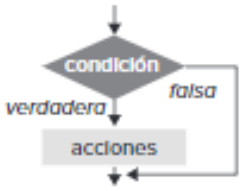
4.4. Alternativa simple (si-entonces/if-then)

La estructura alternativa simple **si-entonces** (en inglés **if-then**) ejecuta una determinada acción cuando se cumple una determinada condición. La selección **si-entonces** evalúa la condición y:

- Si la condición es *verdadera*, entonces ejecuta la acción S1 (o acciones en caso de que S1 sea una acción compuesta y conste de varias acciones).
- Si la condición es *falsa*, entonces no hacer nada.

Las representaciones gráficas de la estructura condicional simple se muestran en la figura 4.4.

a)



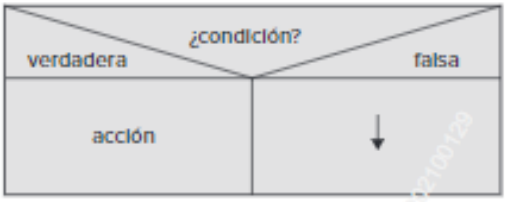
b) Pseudocódigo en español

```
si <condición> entonces
  <acción S1>
fin_si
```

Pseudocódigo en inglés

```
if <condición> then
  <acción S1>
endif
```

c)



Pseudocódigo en español

```
//S1 acción compuesta
si <condición> entonces
  <acción S11>
  <acción S12>
    .
    .
    .
  <acción S1n>
fin_si
```

Figura 4.4. Estructuras alternativas simples: *a*) Diagrama de flujo, *b*) Pseudocódigo, *c*) Diagrama N-S.

Observe que las palabras del pseudocódigo **si** y **fin_si** se alinean verticalmente *indentando* (sangrando) la *<acción>* o bloque de acciones.

Diagrama de sintaxis

Sentencia **if_simple::=**

1. si (<expresión_lógica>)
   inicio
     <sentencia>
   fin

2. si <*expresión lógica*> entonces
     <*Sentencia_compuesta*>
   fin-si

```
Sentencia_compuesta ::=
                    inicio
                         <Sentencias>
                    fin
```

Cristian Pilapanta 1006211-2202100129

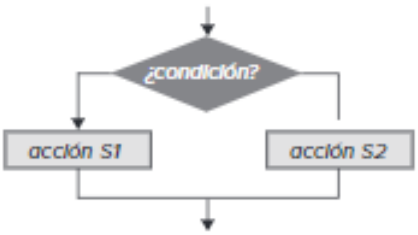
Sintaxis en lenguajes de programación			
Pseudocódigo	**Java**	**C/C++**	**Python**

`si (condición) entonces`
`  acciones`
`fin-si`
	`if (condición) then`
	`begin`
	`  sentencias`
	`end`
		`if (condición)`
		`{`
		`  sentencias`
		`}`
			`If (condición)`
			`sentencias`

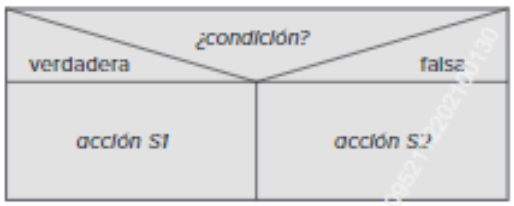
4.4.1. Alternativa doble (`si-entonces-sino/ if-then-else`)

La estructura anterior es muy limitada y normalmente se necesitará una estructura que permita elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición. Si la condición `C` es verdadera, se ejecuta la acción `S1` y, si es falsa, se ejecuta la acción `S2` (figura 4.5).

Cristian Pilapanta 1095211-2202100130



a)



c)

Pseudocódigo en español

```
si <condición> entonces
   <acción S1>
si_no
   <acción S2>
fin_si
```

Pseudocódigo en inglés

```
if <condición> then
   <acción S1>
else
   <acción S2>
endif
```

Pseudocódigo en español

```
//S1 acción compuesta
si <condición> entonces
   <acción S11>
   <acción S12>
   .
   .
   .
   <acción S1n>
si_no
   <acción S21>
   <acción S22>
   .
   .
   .
   <acción S2n>
fin_si
```

b)

Figura 4.5. Estructura alternativa doble: *a*) diagrama de flujo, *b*) pseudocódigo, *c*) diagrama N-S.

Observe que en el pseudocódigo las acciones que dependen de `entonces` y `si_no` están *indentadas* en relación con las palabras `si` y `fin_si`; este procedimiento aumenta la legibilidad de la estructura y es el medio más idóneo para representar algoritmos.

Códigos en Java, C/C++ y Python

Java	C/C++	Python

`if (condición) sentencias`
`else sentencias;`
	`if (condición)`
	`sentencias;`
	`else`
	`sentencias;`
		`if (condición):`
		`else:`
		`sentencias`

EJEMPLO 4.3

Resolución de una ecuación de primer grado.

Si la ecuación es $ax + b = 0$, a y b son los datos, y las posibles soluciones son:

- `a <> 0` `x = -b/a`
- `a = 0 b <> 0` `entonces "solución imposible"`
- `a = 0 b = 0` `entonces "solución indeterminada"`

El algoritmo correspondiente será:

```
algoritmo RESOL1
var
   real : a, b, x
inicio
   leer (a, b)
   si a <> 0 entonces
      x ← -b/a
      escribir(x)
   si_no
      si b <> 0 entonces
         escribir ('solución imposible')
      si_no
         escribir ('solución indeterminada')
      fin_si
   fin_si
fin
```

EJEMPLO 4.4

Calcular la media aritmética de una serie de números positivos.

La media aritmética de n números es:

$$\frac{x1 + x2 + x3 + \ldots + xn}{n}$$

En el problema se supondrá la entrada de datos por el teclado hasta que se introduzca el último número, en nuestro caso -99. Para calcular la media aritmética se necesita saber cuántos números se han introducido hasta llegar a -99; para ello se utilizará un contador n que llevará la cuenta del número de datos introducidos.

Tabla de variables
`real: s` (suma)
`entera: n` (contador de números)
`real: m` (media)

Cristian Pilapanta 1095211-2202100145

```
algoritmo media
var
   real: s, m
   entera: n
inicio
   s ← 0 // inicialización de variables: s y n
   n ← 0
   datos:
      leer (x) // el primer número ha de ser mayor que cero
      si x < 0 entonces ir_a(media)
         si_no
            n ←  n + 1
            s ← s + x
      ir_a(datos)
   fin_si
media:
   m ← s/n // media de los números positivos
   escribir (m)
fin
```

En este ejemplo se observa una bifurcación hacia un punto referenciado por una etiqueta alfanumérica denominada media y otro punto referenciado por datos.

Trate el alumno de simplificar este algoritmo de modo que sólo contenga un punto de bifurcación.

EJEMPLO 4.5

Se desea obtener la nómina semanal –salario neto– de los empleados de una empresa cuyo trabajo se paga por horas y del modo siguiente:

- *Las horas inferiores o iguales a 35 se pagan a una tarifa determinada que se debe introducir por teclado al igual que el número de horas y el nombre del trabajador.*
- *Las horas superiores a 35 se pagarán como extras a un promedio de 1.5 horas normales.*
- *Los impuestos a deducir a los trabajadores varían en función de su sueldo mensual:*
  - *Sueldo < = 2 000, libre de impuestos.*
  - *Los siguientes 220 euros al 20 por 100.*
  - *El resto, al 30 por 100.*

Análisis

Las operaciones a realizar serán:

1. Inicio.
2. Leer nombre, horas trabajadas, tarifa horaria.
3. Verificar si horas trabajadas <= 35, en cuyo caso
   salario_bruto = horas * tarifa; caso contrario,
   salario_bruto = 35 * tarifa + (horas - 35) * tarifa.

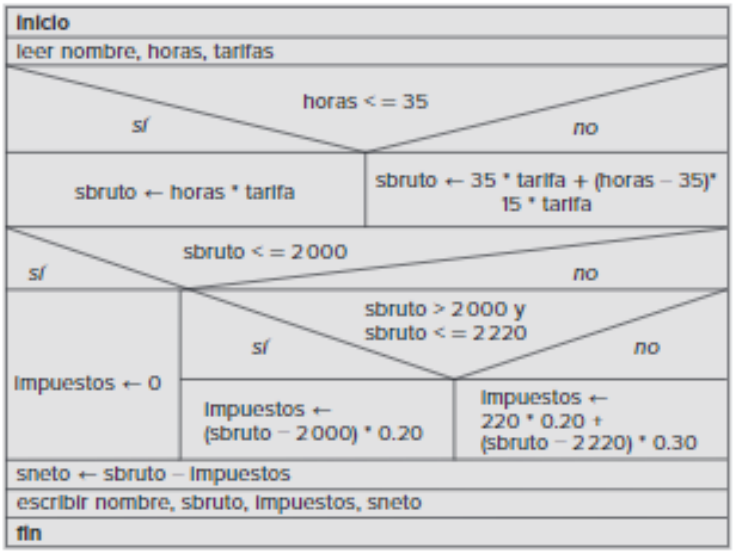
Cristian Pliqarita 1095211-2202100148

```
4. Cálculo de impuestos:
     si salario_bruto <= 2000, entonces impuestos = 0
     si salario_bruto <= 2220 entonces
     impuestos = (salario_bruto - 2000) * 0.20
     si salario_bruto > 2220 entonces
     impuestos = (salario_bruto - 2220) * 0.30 + (220 * 0.20).
5. Cálculo del salario_neto
     salario_neto = salario_bruto - impuestos.
6. Fin.
```

Representación del algoritmo en pseudocódigo

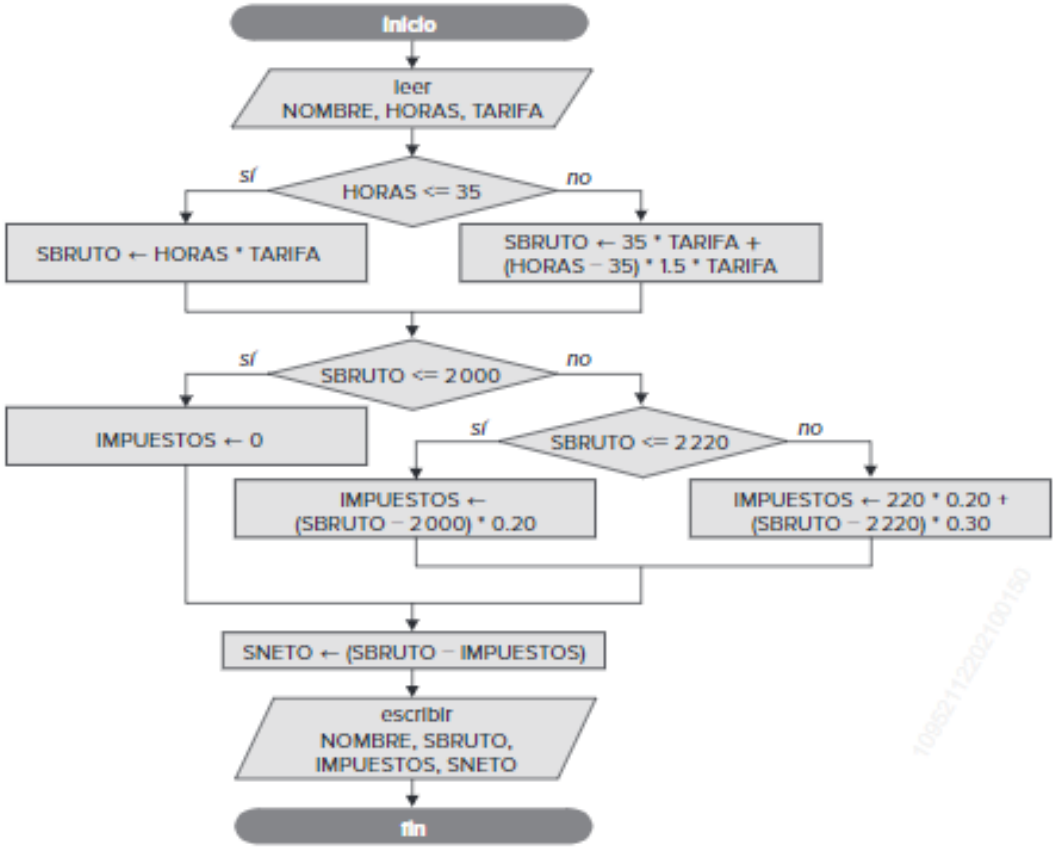
```
algoritmo Nómina
var
   cadena: nombre
   real: horas, impuestos, sbruto, sneto
inicio
   leer(nombre, horas, tarifa)
   si horas <= 35 entonces
      sbruto ← horas * tarifa
   si_no
      sbruto ← 35 * tarifa + (horas - 35) * 1.5 * tarifa
   fin_si
   si sbruto <= 2000 entonces
      impuestos ← 0
   si_no
      si (sbruto > 2000) y (sbruto <= 2220) entonces
         impuestos ← (sbruto - 2000) * 0.20
      si_no
         impuestos ← (220 * 0.20) + (sbruto - 2220)
      fin_si
   fin_si
   sneto ← sbruto - impuestos
   escribir(nombre, sbruto, impuestos, neto)
fin
```

Representación del algoritmo en diagrama N-S



Cristian Palpanta 1095211-2202100149

Representación del algoritmo en diagrama de flujo



Cristian Pilapanta 1095211-2202100150

1095211220210015O

EJEMPLO 4.6

Empleo de estructura selectiva para detectar si un número tiene o no parte fraccionaria.

```
algoritmo Parte_fraccionaria
var
   real: n
inicio
   escribir('Deme número')
   leer(n)
   si n = trunc(n) entonces
      escribir('El número no tiene parte fraccionaria')
   si_no
      escribir('Número con parte fraccionaria')
   fin_si
fin
```

EJEMPLO 4.7

Estructura selectiva para averiguar si un año leído de teclado es o no bisiesto.

```
algoritmo Bisiesto
var
   entero : año
inicio
   leer(año)
   si (año MOD 4 = 0) y (año MOD 100 <> 0) O (año MOD 400 = 0) entonces
      escribir('El año', año, 'es bisiesto')
   si_no
      escribir('El año', año, 'no bisiesto')
   fin_si
fin
```

EJEMPLO 4.8

Algoritmo que nos calcule el área de un triángulo conociendo sus lados. La estructura selectiva se utiliza para el control de la entrada de datos en el programa.

Nota: Área $= \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$ $p = (a+b+c)/2$

```
algoritmo Área_triángulo
var
   real: a, b, c, p, área
inicio
   escribir('Deme los lados')
   leer(a, b, c)
   p ← (a + b + c) / 2
   si (p > a) y (p > b) y (p > c) entonces
      area ← raiz2(p * (p - a) * (p - b) * (p - c))
      escribir(área)
   si_no
      escribir('No es un triángulo')
   fin_si
fin
```

Cristian Pilapanta 1095211-2202100151

1095211220210015

4.5. Alternativa múltiple (según_sea, caso de; switch-case)

Con frecuencia —en la práctica— es necesario que existan más de dos elecciones posibles (por ejemplo, en la resolución de la ecuación de segundo grado existen tres posibles alternativas o caminos a seguir, según que el discriminante sea negativo, nulo o positivo). Este problema, como se verá más adelante, se podría resolver por estructuras alternativas simples o dobles, *anidadas* o *en cascada*; sin embargo, si el número de alternativas es grande este método puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad.

La estructura de decisión múltiple, **según_sea**, **switch**, evaluará una expresión que podrá tomar *n* valores distintos, 1, 2, 3, 4, ..., *n*. Según se elija uno de estos valores en la condición, se realizará una de las *n* acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los *n* posibles.

Los diferentes modelos de pseudocódigo de la estructura de decisión múltiple se representan en las figuras 4.6 y 4.7.

```
Modelo 1:
según_sea expresión (E) hacer
   e1:acción S11
      acción S12
      .
      .
      acción S1a
   e2:acción S21
      acción S22
      .
      .
      acción S2b
   .
   .
   en:acción S31
      acción S32
      .
      .
      acción S3p
   si-no
      acción Sx
fin_según
Modelo 2 (simplificado):
según E hacer
      .
      .
      .
fin_según
```

```
Modelo 3 (simplificado):
opción E de
.
.
fin_opción

Modelo 4 (simplificado):
caso_de E hacer
.
.
.
fin_caso

Modelo 5 (simplificado):
si E es n hacer
.
.
.
fin_si
```

Figura 4.6. Estructuras de decisión múltiple.

```
Modelo 6:
según_sea (expresión) hacer
   caso expresión constante:
      [Sentencia
      sentencia
      ...
      sentencia de ruptura | sentencia ir_a]
   caso expresión constante:
      [Sentencia
      sentencia
      ...
      sentencia de ruptura | sentencia ir_a]
   caso expresión constante:
      [Sentencia
      ...
      sentencia
      sentencia de ruptura | sentencia ir_a]
   [otros:
      [Sentencia
      ...
      sentencia
      sentencia de ruptura | sentencia ir_a]
fin_según
```

Figura 4.7. Sintaxis de sentencia según_sea.

Cristian Pilapanta 1095211-2202100151

```
Sentencia switch (C , C++, Java,  C#)
    switch (expresión)
    {
       case valor1:
          sentencia1;
          sentencia2;
          sentencia3;
            .
            .
       break;
    case valor2:
       sentencia1;
       sentencia2;
       sentencia3;
            .
            .
       break;
            .
            .
    default:
       sentencia1;
       sentencia2;
       sentencia3;
       .
       .
    } // fin de la sentencia compuesta
```

Cristian Pliqueria 1095211-2202100152

1095211222021001152

Diagrama de flujo

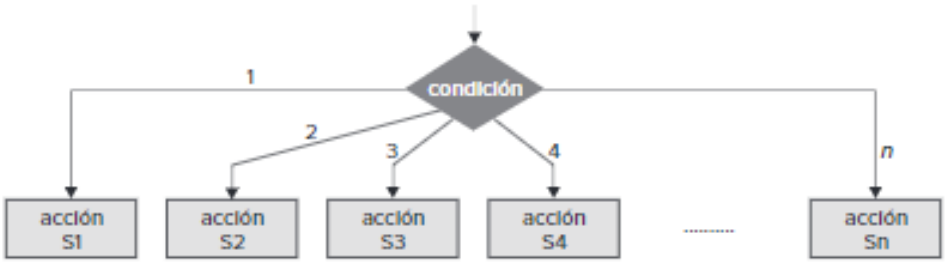
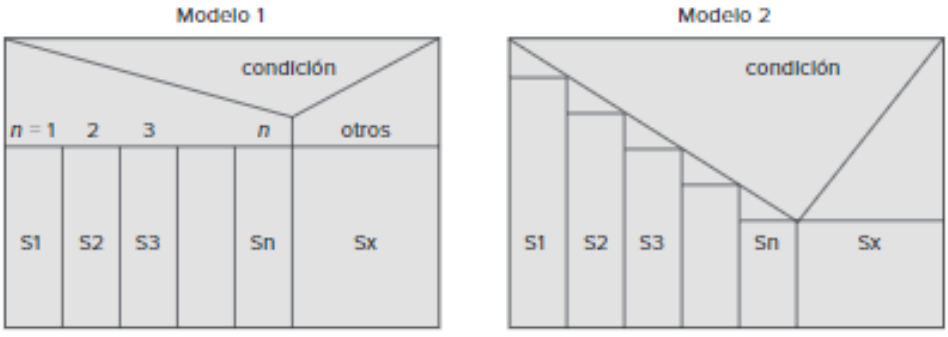


Diagrama N-S



Pseudocódigo

En inglés la estructura de decisión múltiple se representa:

```
case expresión of
   [e1]: acción S1
   [e2]: acción S2
   .
   .
   [en]: acción Sn
   otherwise
      acción Sx
end_case
```

```
case expresión of
   [e1]: acción S1
   [e2]: acción S2


   [en]: acción Sn
   else
      acción Sx
end_case
```

Como se ha visto, la estructura de decisión múltiple en pseudocódigo se puede representar de diversas formas, pudiendo ser las acciones S1, S2, etcétera, *simples* como en el caso anterior o *compuestas* y su funcionalidad varía algo de unos lenguajes a otros.

Notas

1. Observe que para cada valor de la expresión (e) se pueden ejecutar una o varias acciones. Algunos lenguajes como Pascal a estas instrucciones les denominan *compuestas* y las delimitan con las palabras reservadas `begin-end` (`inicio-fin`); es decir, en pseudocódigo.

```
según_sea E hacer
   e1: acción S1
   e2: acción S2
   .
   .
   en: acción Sn
   otros: acción Sx
fin_según
```

o bien en el caso de instrucciones compuestas:

```
según_sea E hacer
   e1: inicio
      acción S11
      acción S12
      .
      .
      acción S1a
      fin
   e2: inicio
      acción S21
      .
      .
      .
      fin
   en: inicio
      .
      .
      .
      fin
   si-no
      acción Sx
fin_según
```

Cristian Pilapanta 1095211-2202100200
10952112202100200

2. Los valores que toman las expresiones (E) no tienen por qué ser consecutivos ni únicos, se pueden considerar rangos de constantes numéricas o de caracteres como valores de la expresión E.

```
caso_de E hacer
   2, 4, 6, 8, 10: escribir ('números pares')
   1, 3, 5, 7, 9: escribir ('números impares')
fin_caso
```

¿Cuál de los modelos expuestos se puede considerar representativo? En realidad, como el pseudocódigo es un lenguaje algorítmico universal, cualesquiera de los modelos se podría ajustar a su presentación; sin embargo, nosotros consideramos como más estándar los modelos 1, 2 y 4. En esta obra seguiremos normalmente el modelo 1, aunque en ocasiones, y para familiarizar al lector en su uso, podremos utilizar los modelos citados 2 y 4.

Los lenguajes como **C** y sus derivados **C++**, **Java** o **C#** utilizan como sentencia selectiva múltiple la sentencia `switch`, cuyo formato es muy parecido al modelo 6.

EJEMPLO 4.9

Se desea diseñar un algoritmo que escriba los nombres de los días de la semana en función del valor de una variable DÍA introducida por teclado.

Los días de la semana son 7; por consiguiente, el rango de valores de DÍA será 1 .. 7, y en caso de que DÍA tome un valor fuera de este rango se deberá producir un mensaje de error advirtiendo la situación anómala.

```
algoritmo DíasSemana
var
   entero: DÍA
inicio
   leer(DÍA)
   según_sea DÍA hacer
   1. : escribir('LUNES')
   2. : escribir('MARTES')
   3. : escribir('MIÉRCOLES')
   4. : escribir('JUEVES')
   5. : escribir('VIERNES')
   6. : escribir('SÁBADO')
   7. : escribir('DOMINGO')
   si-no
      escribir('ERROR')
   fin_según
fin
```

EJEMPLO 4.10

Se desea convertir las calificaciones alfabéticas A, B, C, D, E y F a calificaciones numéricas 4, 5, 6, 7, 8 y 9, respectivamente.

Los valores de A, B, C, D, E y F se representarán por la variable LETRA, el algoritmo de resolución del problema es:

```
algoritmo Calificaciones
var
   carácter: LETRA
```

Cristian Palpanta 1095211-2202100201

```
   entero: calificación
inicio
   leer(LETRA)
   según_sea LETRA hacer
      'A': calificación ← 4
      'B': calificación ← 5
      'C': calificación ← 6
      'D': calificación ← 7
      'E': calificación ← 8
      'F': calificación ← 9
   otros:
      escribir ('ERROR')
   fin_según
fin
```

Como se ve en el pseudocódigo, no se contemplan otras posibles calificaciones —por ejemplo, 0, —; si así fuese, habría que modificarlo en el siguiente sentido:

```
según_sea LETRA hacer
      'A': calificación ← 4
      'B': calificación ← 5
      'C': calificación ← 6
      'D': calificación ← 7
      'E': calificación ← 8
      'F': calificación ← 9
   otros: calificación ← 0
fin_según
```

Cristian Palpanta 1095211-2202100201

1095211220210020

EJEMPLO 4.11

Se desea leer por teclado un número comprendido entre 1 y 10 (inclusive) y se desea visualizar si el número es par o impar.

En primer lugar, se deberá detectar si el número está comprendido en el rango válido (1 a 10) y a continuación si el número es 1, 3, 5, 7, 9, escribir un mensaje *impar*; si es 2, 4, 6, 8, 10, escribir un mensaje de *par*.

```
algoritmo PAR_IMPAR
var entero: número
inicio
    leer(número)
    si número >= 1 y número <= 10 entonces
       según_sea número hacer
          1, 3, 5, 7, 9: escribir ('impar')
          2, 4, 6, 8, 10: escribir ('par')
       fin_según
    fin_si
fin
```

EJEMPLO 4.12

Leída una fecha, decir el día de la semana, suponiendo que el día 1 de dicho mes fue lunes.

```
algoritmo Día_semana
var
   entero: día
```

```
inicio
   escribir('Diga el día')
   leer(día)
   según_sea día MOD 7 hacer
      1:
         escribir('Lunes')
      2:
         escribir('Martes')
      3:
         escribir('Miércoles')
      4:
         escribir('Jueves')
      5:
         escribir('Viernes')
      6:
         escribir('Sábado')
      0:
         escribir('Domingo')
   fin_según
fin
```

Cristian Pilapanta 1095211-2202100203

EJEMPLO 4.13

Preguntar qué día de la semana fue el día 1 del mes actual y calcular que día de la semana es hoy.

```
algoritmo Día_semana_modificado
var
   entero: día,d1
   carácter: día1
inicio
   escribir('El día 1 fue (L,M,X,J,V,S,D)')
   leer(día1)
   según_sea día1 hacer
      'L':
         d1 ← 0
      'M':
         d1 ← 1
      'X':
         d1 ← 2
      'J':
         d1 ← 3
      'V':
         d1 ← 4
      'S':
         d1 ← 5
      'D':
         d1 ← 6
      si_no
         d1 ← -40
fin_según
escribir('Diga el día')
```

```
leer(día)
día ← día + d1
según_sea día MOD 7 hacer
   1:
      escribir('Lunes')
   2:
      escribir('Martes')
   3:
      escribir('Miércoles')
   4:
      escribir('Jueves')
   5:
      escribir('Viernes')
   6:
      escribir('Sábado')
   0:
      escribir('Domingo')
   fin_según
fin
```

Cristian Pilapanta 1006211-2202100206

EJEMPLO 4.14

Algoritmo que nos indique si un número entero, leído del teclado, tiene 1, 2, 3 o más de 3 dígitos. Considerar los negativos.

Se puede observar que la estructura **según_sea** *<expresión>* **hacer** son varios **si** *<expr. lógica>* **entonces ...** anidados en la rama **si_no**. Si se cumple el primero ya no pasa por los demás.

1006211-2202100206

```
algoritmo Dígitos
var
   entero : n
inicio
   leer(n)
   según_sea n hacer
      -9 .. 9:
         escribir('Tiene 1 dígito')
      -99 .. 99:
         escribir('Tiene 2')
      -999 .. 999:
         escribir('Tiene tres')
   si_no
      escribir('Tiene más de tres')
   fin_según
fin
```

4.6. Estructuras de decisión anidadas (en escalera)

Las estructuras de selección **si-entonces** y **si-entonces-si_no** implican la selección de una de dos alternativas. Es posible también utilizar la instrucción **si** para diseñar estructuras de selección que contengan más de dos alternativas. Por ejemplo, una estructura **si-entonces** puede contener otra estructura **si-entonces**, y esta estructura **si-entonces** puede contener otra, y así sucesivamente cualquier número de veces; a su vez, dentro de cada estructura pueden existir diferentes acciones.

Las estructuras `si` interiores a otras estructuras `si` se denominan *anidadas* o *encajadas*:

```
si <condición1> entonces
   si <condición2> entonces
   .
   .
   .
      <acciones>
   fin_si
fin_si
```

Una estructura de selección de *n* alternativas o de decisión múltiple puede ser construida utilizando una estructura `si` con este formato:

```
si <condición1> entonces
   <acciones>
si_no
   si <condición2> entonces
      <acciones>
   si_no
      si <condición3> entonces
         <acciones>
      si_no
      .
      .
      .
      fin_si
   fin_si
fin_si
```

Una estructura selectiva múltiple constará de una serie de estructuras `si`, unas interiores a otras. Como las estructuras `si` pueden volverse bastante complejas para que el algoritmo sea claro, será preciso utilizar *indentación* (sangría o sangrado), de modo que exista una correspondencia entre las palabras reservadas `si` y `fin_si`, por un lado, y `entonces` y `si_no`, por otro.

La escritura de las estructuras puede variar de unos lenguajes a otros, por ejemplo, una estructura `si` admite también los siguientes formatos:

```
si <expresión booleana1> entonces
   <acciones>
si_no
   si <expresión booleana2> entonces
      <acciones>
   si_no
      si <expresión booleana3> entonces
         <acciones>
      si_no
         <acciones>
      fin_si
   fin_si
fin_si
```

Cristian Pilapanta 1006211-2202102332

o bien

```
si <expresión booleana1> entonces
   <acciones>
si_no si <expresión booleana2> entonces
   <acciones>
   fin_si
.
.
.
fin_si
```

EJEMPLO 4.15

Diseñar un algoritmo que lea tres números A, B, C *y visualice en pantalla el valor del más grande. Se supone que los tres valores son diferentes.*

Los tres números son A, B y C; para calcular el más grande se realizarán comparaciones sucesivas por parejas.

```
algoritmo Mayor
var
   real: A, B, C, Mayor
inicio
   leer(A, B, C)
   si A > B entonces
      si A > C entonces
         Mayor ← A          //A > B, A > C
      si_no
         Mayor ← C          //C >= A > B
      fin_si
   si_no
      si B > C entonces
         Mayor ← B          //B >= A, B > C
      si_no
         Mayor ← C          //C >= B >= A
      fin_si
   fin_si
   escribir('Mayor:', Mayor)
fin
```

EJEMPLO 4.16

El siguiente algoritmo lee tres números diferentes, A, B, C *e imprime los valores máximo y mínimo. El procedimiento consistirá en comparaciones sucesivas de parejas de números.*

```
algoritmo Ordenar
var
   real: a,b,c
inicio
   escribir('Deme 3 números')
   leer(a, b, c)
   si a > b entonces          // consideramos los dos primeros (a, b)
```

Cristian Palpanta 1095211-2202102332

```
                              // y los ordenamos
      si b > c entonces       // tomo el 3º (c) y lo comparo con el menor
                              // (a o b)
         escribir(a, b, c)
      si-no                   // si el 3º es mayor que el menor averiguo si
      si c > a entonces       // va delante o detrás del mayor
         escribir(c, a, b)
         si_no
            escribir(a, c, b)
         fin_si
      fin_si
   si_no
      si a > c entonces
         escribir(b, a, c)
      si_no
         si c > b entonces
            escribir(c, b, a)
         si_no
            escribir(b, c, a)
         fin_si
      fin_si
   fin_si
fin
```

Cristian Pilapanta 1095211-2202102332

1095211-2202102332

EJEMPLO 4.17

Pseudocódigo que nos permita calcular las soluciones de una ecuación de segundo grado, incluyendo los valores imaginarios.

```
algoritmo Soluciones_ecuación
var
   real: a,b,c,d,x1,x2,r,i
inicio
   escribir('Deme los coeficientes')
   leer(a, b, c)
   si a = 0 entonces
      escribir('No es ecuación de segundo grado')
   si_no
      d ← b * b - 4 * a * c
      si d = 0 entonces
         x1 ← -b / (2 * a)
         x2 ← x1
         escribir(x1, x2)
      si_no
         si d > 0 entonces
            x1 ← (-b + raiz2(d)) / (2 * a)
            x2 ← (-b - raiz2(d)) / (2 * a)
            escribir(x1, x2)
         si_no
```

```
            r ← (-b) / (2 * a)
            i ← raiz2(abs(d)) / (2 * a)
            escribir(r, '+', i, 'i')
            escribir(r, '-', i, 'i')
         fin_si
      fin_si
   fin_si
fin
```

EJEMPLO 4.18

Algoritmo al que le damos la hora HH, MM, SS *y nos calcule la hora dentro de un segundo. Leeremos las horas, minutos y segundos como números enteros.*

```
algoritmo Hora_segundo_siguiente
var
   entero : hh, mm, ss
inicio
   escribir('Deme hh,mm,ss')
   leer(hh, mm, ss)
   si (hh < 24) y (mm < 60) y (ss < 60) entonces
      ss ← ss + 1
      si ss = 60 entonces
         ss ← 0
         mm ← mm + 1
         si mm = 60 entonces
            mm ← 0
            hh ← hh + 1
            si hh = 24 entonces
               hh ← 0
            fin_si
         fin_si
      fin_si
         escribir(hh, ':', mm, ':', ss)
   fin_si
fin
```

Cristian Pilapanta 1095211-2202102341

4.7. La sentencia ir-a (goto) [no recomendable su uso, con excepciones]

El flujo de control de un algoritmo es siempre secuencial, excepto cuando las estructuras de control estudiadas anteriormente realizan transferencias de control no secuenciales.

La programación estructurada permite realizar programas fáciles y legibles utilizando las tres estructuras ya conocidas: *secuenciales, selectivas* y *repetitivas*. Sin embargo, en ocasiones es necesario realizar bifurcaciones incondicionales, para ello se recurre a la instrucción `ir_a` (`goto`). Esta instrucción siempre ha sido problemática y prestigiosos informáticos, como Dijkstra, han tachado la instrucción `goto` como nefasta y perjudicial para los programadores, y recomiendan no utilizarla en sus algoritmos y programas. Por ello, la mayoría de los lenguajes de

programación, desde el mítico Pascal –padre de la programación estructurada– como los lenguajes más utilizados en los últimos años y en la actualidad como **C**, **C++**, **C♯**, *huyen* de esta instrucción y prácticamente no la utilizan nunca, aunque eso sí, mantienen en su juego de sentencias esta "dañina" sentencia por si en situaciones excepcionales es necesario recurrir a ella. **Java** no tiene la sentencia, aunque sí mantiene la palabra reservada `goto`; sin embargo, **Python** no incorpora la sentencia ni mantiene la palabra reservada `goto`.

> La sentencia `ir_a` (`goto`) es la forma de control más primitiva en los programas de computadoras y corresponde a una bifurcación **incondicional en código máquina. Aunque en lenguajes modernos como VB .NET (Visual Basic .NET), C, C++ y C#** prácticamente no se utiliza. Otros lenguajes modernos como Java no contienen la sentencia `goto`, aunque sí es una palabra reservada. Python no contiene la sentencia ni la palabra reservada `goto`.

En general, no existe ninguna necesidad de utilizar instrucciones `ir_a`. Cualquier algoritmo o programa que se escriba con instrucciones `ir_a` se puede reescribir para hacer lo mismo y no incluir ninguna instrucción `ir_a`. Un programa que utiliza muchas instrucciones `ir_a` es más difícil de leer que un programa bien escrito que utiliza pocas o ninguna instrucción `ir_a`.

En muy pocas situaciones las instrucciones `ir_a` son útiles; tal vez, las únicas razonables son diferentes tipos de situaciones de salida de bucles. Cuando un error u otra condición de terminación se encuentra una instrucción `ir_a` puede ser utilizada para saltar directamente al final de un bucle, subprograma o un procedimiento completo.

Las bifurcaciones o *saltos* producidos por una instrucción `ir_a` deben realizarse a instrucciones que estén numeradas o posean una etiqueta que sirva de punto de referencia para el salto. Por ejemplo, un programa puede ser diseñado para terminar con una detección de un error.

```
algoritmo error
.
.
.
   si <condición error> entonces
      ir_a(100)
   fin_si
100: fin
```

La sentencia `ir-a` (`goto`) o sentencia de invocación directa transfiere el control del programa a una posición especificada por el programador. En consecuencia, interfiere con la ejecución secuencial de un programa. La sentencia `ir-a` tiene una historia muy controvertida y a la que se ha hecho merecedora por las malas prácticas de enseñanza que ha producido. Se puede decir que es un elemento superfluo del lenguaje y sólo en muy contadas ocasiones, precisamente con la sentencia `switch` (en algunas aplicaciones muy concretas) podría tener alguna utilidad práctica.

Como regla general, es interesante que sepa cómo funciona esta sentencia, pero *no la utilice nunca a menos que le sirva en un momento determinado para resolver una situación no prevista y que un salto prefijado le ayude en esa resolución*. La sintaxis de la sentencia `ir_a` tiene tres variantes:

`ir_a` *`etiqueta`*	(`goto` *`etiqueta`*)
`ir_a` *`caso`*	(`goto case`, en la sentencia `switch`)
`ir_a` *`otros`*	(`goto default`, en la sentencia `switch`)

La construcción `ir_a` etiqueta consta de una sentencia `ir_a` y una sentencia asociada con una etiqueta. Cuando se ejecuta una sentencia `ir_a`, se transfiere el control del programa a la etiqueta asociada, como se ilustra en el siguiente recuadro.

Cristian Pilapanta 1095211-2202102342

```
...
inicio
   ...
   ir_a etiqueta1
   ...
fin
   ...
etiqueta1:
...          // el flujo del programa salta a la sentencia siguiente
             // a la rotulada por etiqueta1
```

Por lo general, en el caso de soportar la sentencia `ir_a` como el lenguaje **C♯**, esta sentencia transfiere el control fuera, no dentro, de un ámbito anidado. Por lo tanto, la siguiente sentencia no es válida.

```
inicio
       ir_a etiquetaC
   ...
   inicio                          No válido: transferencia de control dentro
      ...                          de un ámbito anidado.
      etiquetaC
      ...
   fin
   ...
fin
```

Sin embargo, el siguiente código sí se suele aceptar por el compilador *(en concreto C♯)*:

```
inicio
   ...
   inicio
      ...
      ir_a etiquetaC
      ...
   fin
   etiquetaC
   ...
fin
```

La sentencia `ir_a` pertenece a un grupo conocidas como **sentencias de salto** *(jump)*; que hacen que el flujo de control salte a otra parte del programa. Otras sentencias de salto o bifurcación que se encuentran en los lenguajes de programación, tanto tradicionales como nuevos (**C**, **C++**, **C♯**, **Java**, **Pascal**) son `interrumpir` (`break`), `continuar` (`continue`), `volver` (`return`) y `lanzar` (`throw`). Las tres primeras suelen utilizarse con sentencias de control y como retorno de ejecución de funciones o métodos. La sentencia `throw` puede utilizarse en los lenguajes de programación que poseen mecanismos de manipulación de excepciones, tales como los lenguajes orientados a objetos **C++**, **Java** y **C♯**.

Cristian P Alpanta 1095211-2202102343

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

4.1. *Leer dos números y deducir si están en orden creciente.*

Solución

Dos números *a* y *b* están en orden creciente si $a <= b$.

```
algoritmo comparación1
var
   real : a, b
inicio
   escribir('dar dos números')
   leer(a, b)
   si a <= b entonces
      escribir('orden creciente')
   si_no
      escribir('orden decreciente')
   fin_si
fin
```

4.2. *Determinar el precio del billete de ida y vuelta en avión, conociendo la distancia a recorrer y sabiendo que si el número de días de estancia es superior a 7 y la distancia superior a 800 km el billete tiene una reducción del 30 por 100. El precio por km es de 2.5 euros.*

Solución

Análisis

Las operaciones secuenciales a realizar son:

1. Leer distancia, duración de la estancia y precio del kilómetro.
2. Comprobar si distancia > 800 km y duración > 7 días.
3. Cálculo del precio total del billete: precio total = distancia * 2.5

   - si distancia > 800 km. y duración > 7 días
     precio total = (distancia * 2.5) - 30/100 * (precio total).

Pseudocódigo

```
algoritmo billete
var
   entero: E
   real: D, PT
inicio
   leer(E)
   PT ← 2.5 * D
   si (D > 800) y (E > 7) entonces
      PT ← PT - PT * 30/100
   fin_si
   escribir('Precio del billete', PT)
fin
```

4.3. *Los empleados de una fábrica trabajan en dos turnos: diurno y nocturno. Se desea calcular el jornal diario de acuerdo con los siguientes puntos:*

1. *La tarifa de las horas diurnas es de 5 euros.*
2. *La tarifa de las horas nocturnas es de 8 euros.*
3. *En caso de ser domingo, la tarifa se incrementará en 2 euros el turno diurno y 3 euros el turno nocturno.*

Cristian Palpanta 1095211-2202102344

Solución

Análisis

El procedimiento a seguir es:

1. Leer nombre del turno, horas trabajadas (HT) y día de la semana.
2. Si el turno es nocturno, aplicar la fórmula JORNAL = 8 * HT.
3. Si el turno es diurno, aplicar la fórmula JORNAL = 5 * HT.
4. Si el día es domingo:
   - *turno diurno* JORNAL = (5 + 2) * ht,
   - *turno nocturno* JORNAL = (8 + 3) * HT.

Pseudocódigo

```
algoritmo jornal
var
   cadena : Día, Turno
   real : HT, Jornal
inicio
   leer(HT, Día, Turno)
   si Día < > 'Domingo' entonces
      si Turno = 'diurno' entonces
         Jornal ← 5 * HT
      si_no
         Jornal ← 8 * HT
      fin_si
   si_no
      si Turno = 'diurno' entonces
         Jornal ← 7 * HT
      si_no
         Jornal ← 11 * HT
      fin_si
   fin_si
   escribir(Jornal)
fin
```

4.4. *Construir un algoritmo que escriba los nombres de los días de la semana, en función de la entrada correspondiente a la variable DÍA.*

Solución

Análisis

El método a seguir consistirá en clasificar cada día de la semana con un número de orden:

1. LUNES
2. MARTES
3. MIÉRCOLES
4. JUEVES
5. VIERNES
6. SÁBADO
7. DOMINGO

si Día > 7 y < 1 **error de entrada. rango (1 a 7).**

si el lenguaje de programación soporta sólo la estructura **si-entonces-si_no (if-then-else)**, se codifica con el método 1; caso de soportar la estructura **según_sea (case)**, la codificación será el método 2.

Cristian Pilapanta 1095211-2202102345
10952112202102345

Pseudocódigo

Método 1

```
algoritmo Días_semana1
var
   entero: Día
inicio
   leer(Día)
   si Día = 1 entonces
      escribir('LUNES')
   si_no
      si Día = 2 entonces
         escribir('MARTES')
      si_no
         si Día = 3 entonces
            escribir('MIÉRCOLES')
         si_no
            si Día = 4 entonces
               escribir('JUEVES')
            si_no
               si Día = 5 entonces
                  escribir('VIERNES')
               si_no
                  si Día = 6 entonces
                     escribir('SÁBADO')
                  si_no
                     si Día = 7 entonces
                        escribir('DOMINGO')
                     si_no
                        escribir('error')
                        escribir('rango 1-7')
                     fin_si
                  fin_si
               fin_si
            fin_si
         fin_si
      fin_si
   fin_si
fin
```

Método 2

```
algoritmo Días_semana2
var
   entero: Día
inicio
   leer(Día)
   segun_sea Día hacer
      1: escribir('LUNES')
      2: escribir('MARTES')
      3: escribir('MIÉRCOLES')
      4: escribir('JUEVES')
      5: escribir('VIERNES')
      6: escribir('SÁBADO')
      7: escribir('DOMINGO')
      en_otro_caso escribir('error de entrada, rango 1-7')
   fin_según
fin
```

Cristian Palpanta 1065211-2202102345

CONCEPTOS CLAVE

- Ámbito
- Claúsula `else`
- Condición
- Condición falsa
- Condición verdadera
- Expresión *booleana*
- Expresión lógica
- Operador de comparación
- Operador de relación
- Operador lógico
- Sentencia compuesta
- Sentencia `if`, `switch`
- Sentencia `según-sea`
- Sentencia `si-entonces`
- Sentencia `si-entonces-sino`
- `Si` anidada
- `Si` en escalera

RESUMEN

Las estructuras de selección `si` y `según_sea` son sentencias de bifurcación que se ejecutan en función de sus elementos relacionados en las expresiones o condiciones correspondientes que se forman con operadores lógicos y de comparación. Estas sentencias permiten escribir algoritmos que realizan tomas de decisiones y reaccionan de modos diferentes a datos diferentes.

1. Una sentencia de bifurcación es una construcción del lenguaje que utiliza una condición dada (expresión *booleana*) para decidir entre dos o más direcciones alternativas (ramas o bifurcaciones) a seguir en un algoritmo.
2. Un programa sin ninguna sentencia de bifurcación o iteración se ejecuta secuencialmente, en el orden en que están escritas las sentencias en el código fuente o algoritmo. Estas sentencias se denominan secuenciales.
3. La sentencia `si` es la sentencia de decisión o selectiva fundamental. Contiene una expresión *booleana* que controla si se ejecuta una sentencia (simple o compuesta).
4. Combinando una sentencia `si` con una cláusula `sino`, el algoritmo puede elegir entre la ejecución de una o dos acciones alternativas (simple o compuesta).
5. Las expresiones relacionales, también denominadas *condiciones simples,* se utilizan para comparar operandos. Si una expresión relacional es verdadera, el valor de la expresión se considera en los lenguajes de programación el entero 1. Si la expresión relacional es falsa, entonces toma el valor entero de 0.
6. Se pueden construir condiciones complejas utilizando expresiones relacionales mediante los operadores lógicos, `Y`, `O`, `NO`.
7. Una sentencia `si-entonces` se utiliza para seleccionar entre dos sentencias alternativas basadas en el valor de una expresión. Aunque las expresiones relacionales se utilizan normalmente para la expresión a comprobar, se puede utilizar cualquier expresión válida. Si la expresión (condición) es verdadera se ejecuta la `sentencia1` y en caso contrario se ejecuta la `sentencia2`.

```
si (expresión) entonces
      sentencia1
sino
      sentencia2
fin_si
```

8. Una sentencia compuesta consta de cualquier número de sentencias individuales encerradas dentro de las palabras reservadas `inicio` y `fin` (en el caso de lenguajes de programación como C y C++, entre una pareja de llaves { y }). Las sentencias compuestas se tratan como si fuesen una única unidad y se pueden utilizar en cualquier parte en que se utilice una sentencia simple.
9. Anidando sentencias `si`, unas dentro de otras, se pueden diseñar construcciones que pueden elegir entre ejecutar cualquier número de acciones (sentencias) diferentes (simples o compuestas).
10. La sentencia `según_sea` es una sentencia de selección múltiple. El formato general de una sentencia `según_sea` (`switch`, en inglés) es:

```
según_sea E hacer
   el: inicio
      acción S11
      acción S12
      .
      .
      acción S1a
```

Cristian Pilapanta 1096211-2202102346

```
        fin
    e2: inicio
        acción S21
        .
        .
        .
        fin
    en: inicio
        .
        .
        .
        fin
    otros: acción Sx
fin_según
```

El valor de la expresión entera se compara con cada una de las constantes enteras (también pueden ser carácter o expresiones constantes). La ejecución del programa se transfiere a la primera sentencia compuesta cuya etiqueta precedente (valor `e1`, `e2`,--) coincida con el valor de esa expresión y continúa su ejecución hasta la última sentencia de ese bloque, y a continuación termina la sentencia **`según_sea`**. En caso de que el valor de la expresión no coincida con ningún valor de la lista, entonces se realizan las sentencias que vienen a continuación de la cláusula "otros".

11. La sentencia **`ir_a`** (**`goto`**) transfiere el control (salta) a otra parte del programa y, por consiguiente, pertenece al grupo de sentencias denominadas de salto o bifurcación. Es una sentencia muy controvertida y propensa a errores, por lo que su uso es muy reducido, por no decir nunca, y sólo se recomienda en una sentencia **`según_sea`** para salir del correspondiente bloque de sentencias.

12. La sentencia **`según_sea`** (**`switch`**) está construida a medida de los requisitos del programador para seleccionar múltiples sentencias (simples o compuestas) y es similar a múltiples sentencias **`si-entonces`** anidadas pero con un rango de aplicaciones más restringido. Por lo general, es más recomendable usar sentencias **`según_sea`** que sentencias **`si-entonces`** anidadas porque ofrecen un código más simple, más claro y más eficiente.

Cristian Palapanta 10052112202102347

10052112202102347

EJERCICIOS

4.1. Escribir las sentencias **`si`** apropiadas para cada una de las siguientes condiciones:

a) Si un ángulo es igual a 90°, imprimir el mensaje "`El ángulo es recto`" sino imprimir el mensaje "`El ángulo no es recto`".
b) Si la temperatura es superior a 100 °C, visualizar el mensaje "por encima del punto de ebullición del agua" sino visualizar el mensaje "por debajo del punto de ebullición del agua".
c) Si el número es positivo, sumar el numeral a total de positivos, sino sumar al total de negativos.
d) Si `x` es mayor que `y`, y `z` es menor que 20, leer un valor para `p`.
e) Si `distancia` es mayor que 20 y menos que 35, leer un valor para `tiempo`.

4.2. Escribir un programa que solicite al usuario introducir dos números. Si el primer número introducido es mayor que el segundo número, el programa debe imprimir el mensaje `El primer número es el mayor`, en caso contrario el programa debe imprimir el mensaje `El primer número es el más pequeño`. Considerar el caso de que ambos números sean iguales e imprimir el correspondiente mensaje.

4.3. Dados tres números deducir cuál es el central.

4.4. Calcular la raíz cuadrada de un número y escribir su resultado. Considerando el caso en que el número sea negativo.

4.5. Escribir los diferentes métodos para deducir si una variable o expresión numérica es par.

4.6. Diseñar un programa en el que a partir de una fecha introducida por teclado con el formato `DÍA`, `MES`, `AÑO` se obtenga la fecha del día siguiente.

4.7. Se desea realizar una estadística de los pesos de los alumnos de un colegio de acuerdo con la siguiente tabla:

Alumnos de menos de 40 kg. Alumnos entre 40 y 50 kg.
Alumnos de más de 50 kg y menos de 60 kg.
Alumnos de más o igual a 60 kg.

4.8. Realizar un algoritmo que averigüe si dados dos números introducidos por teclado uno es divisor del otro.

4.9. Un ángulo se considera agudo si es menor de 90º, obtuso si es mayor de 90º y recto si es igual a 90º. Utilizando esta información, escribir un algoritmo que acepte un ángulo en grados y visualice el tipo de ángulo correspondiente a los grados introducidos.

4.10. El sistema de calificación estadounidense suele calcularse de acuerdo con el siguiente cuadro:

Grado numérico	Grado en letra
Grado mayor o igual a 90	A
Menor de 90 pero mayor o igual a 80	B
Menor de 80 pero mayor o igual a 70	C
Menor de 70 pero mayor o igual a 69	D
Menor de 69	F

Utilizando esta información, escribir un algoritmo que acepte una calificación numérica del estudiante (0-100), convierta esta calificación a su equivalente en letra y visualice la calificación correspondiente en letra.

4.11. Escribir un programa que seleccione la operación aritmética a ejecutar entre dos números dependiendo del valor de una variable denominada `selecciónOp`.

4.12. Escribir un programa que acepte dos números reales de un usuario y un código de selección. Si el código introducido de selección es 1, entonces el programa suma los dos números introducidos previamente y se visualiza el resultado; si el código de selección es 2, los números deben ser multiplicados y visualizado el resultado, y si el código seleccionado es 3, el primer número se debe dividir por el segundo número y visualizarse el resultado.

4.13. Escribir un algoritmo que visualice el siguiente doble mensaje

```
Introduzca un mes (1 para Enero, 2 para
Febrero,…)
Introduzca un día del mes
```

El algoritmo acepta y almacena un número en la variable `mes` en respuesta a la primera pregunta y acepta y almacena un número en la variable `día` en respuesta a la segunda pregunta. Si el mes introducido no está entre 1 y 12 se debe visualizar un mensaje de información al usuario advirtiéndole que el número introducido no es válido como mes; de igual forma se procede con el número que representa el día del mes si no está en el rango entre 1 y 31.

Modifique el algoritmo para prever que el usuario introduzca números con decimales.

Nota: como los años bisiestos, febrero tiene 29 días, modifique el programa de modo que advierta al usuario si introduce un día de mes que no existe (por ejemplo, 30 o 31). Considere también el hecho de que hay meses de 30 días y otros de 31 días, de modo que nunca se produzca error de introducción de datos o que en su defecto se visualice un mensaje al usuario advirtiéndole del error cometido.

4.14. Escriba un programa que simule el funcionamiento normal de un ascensor (elevador) moderno con 25 pisos (niveles) y que posee dos botones de *SUBIR* y *BAJAR*, excepto en el piso (nivel) inferior, que sólo existe botón de llamada para *SUBIR* y en el último piso (nivel) que sólo existe botón de *BAJAR*.

Cristian Pilapanta 1005211-2202102348

CAPÍTULO 5

Flujo de control II: estructuras repetitivas

5.1. Estructuras repetitivas
5.2. Estructura `mientras ("while")`
5.3. Estructura `hacer-mientras ("do-while")`
5.4. Diferencias entre `mientras (while)` y `hacer-mientras (do-while)`: una aplicación en C++
5.5. Estructura `repetir ("repeat")`
5.6. Estructura `desde/para ("for")`
5.7. Salidas internas de los bucles
5.8. Sentencias de salto `interrumpir (break)` y `continuar (continue)`
5.9. Comparación de bucles `while`, `for` y `do-while`: una aplicación en C++
5.10. Diseño de bucles (lazos)
5.11. Estructuras repetitivas anidadas
5.12. Sentencias de salto en C++, Java y Python: `break`, `continue`, `return`, `goto`

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS
REFERENCIAS BIBLIOGRÁFICAS

Cristian Pilapanta 1096211-2202102349

INTRODUCCIÓN

Los programas utilizados hasta este momento han examinado conceptos de programación, tales como entradas, salidas, asignaciones, expresiones y operaciones, sentencias secuenciales y de selección. Sin embargo, muchos problemas requieren de características de repetición, en las que algunos cálculos o secuencia de instrucciones se repiten una y otra vez, utilizando diferentes conjuntos de datos. Ejemplos de tales tareas repetitivas incluyen verificaciones (chequeos) de entradas de datos de usuarios hasta que se introduce una entrada aceptable, como una contraseña válida; conteo y acumulación de totales parciales; aceptación constante de entradas de datos y recálculos de valores de salida, cuyo proceso sólo se detiene cuando se introduce o se presenta un valor centinela.

Este capítulo examina los diferentes métodos que utilizan los programadores para construir secciones de código repetitivas. Se describe y analiza el concepto de **bucle** como la sección de código que se repite y que se denomina así ya que cuando termina la ejecución de la última sentencia, el flujo de control vuelve a la primera sentencia y comienza otra repetición de las sentencias del código. Cada repetición se conoce como *iteración* o *pasada a través del bucle*.

Se estudian los bucles más típicos: **`mientras`**, **`hacer-mientras`**, **`repetir-hasta que`** y **`desde`** (o **`para`**).

5.1. Estructuras repetitivas

Las computadoras están diseñadas especialmente para todas aquellas aplicaciones en las cuales una operación o conjunto de ellas debe repetirse muchas veces. Un tipo muy importante de estructura es el algoritmo, necesario para repetir una o varias acciones un número determinado de veces. Un programa que lee una lista de números puede repetir la misma secuencia de mensajes al usuario e instrucciones de lectura hasta que todos los números de un fichero (o carpeta) se lean.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se conocen *bucles o lazos*, y se denomina *iteración* al hecho de repetir la ejecución de una secuencia de acciones. Para aclarar la situación, veamos un ejemplo, supongamos que se desea sumar una lista de números escritos desde teclado, por ejemplo, calificaciones de los alumnos de una clase. El medio conocido hasta ahora es leer los números y añadir sus valores a una variable SUMA que contenga el resultado parcial. La variable SUMA se hace igual a cero y, a continuación, se incrementa en el valor del número cada vez que uno de ellos se lea. El algoritmo que resuelve este problema es:

```
algoritmo suma
var
   entero : SUMA, NÚMERO
inicio
   SUMA ← 0
   leer(número)
   SUMA ← SUMA + número
   leer(número)
   SUMA ← SUMA + número
   leer(número)
fin
```

y así sucesivamente para cada número de la lista. En otras palabras, el algoritmo repite muchas veces las acciones.

```
leer(número)
SUMA ← SUMA + número
```

Las dos principales preguntas a realizarse en el diseño de un bucle son: ¿qué contiene el bucle? y ¿cuántas veces debe repetirse?

Cuando se utiliza un bucle para sumar una lista de números, es necesario saber cuántos números se sumarán. Para ello debemos conocer algún medio para *detener* el bucle. En el ejemplo anterior usaremos la técnica de solicitar al usuario el número que desea, por ejemplo, N. Existen dos procedimientos para contar el número de iteraciones, uno es usar una variable TOTAL que se inicializa a la cantidad de números que se desea y a continuación se decrementa en uno cada vez que el bucle se repite (este procedimiento añade una acción más al cuerpo del bucle: TOTAL ← TOTAL - 1), o bien inicializar la variable TOTAL en 0 o en 1, e ir incrementando en uno a cada iteración hasta llegar al número deseado.

```
algoritmo suma_número
var
   entero : N, TOTAL
   real : NÚMERO, SUMA
inicio
   leer(N)
   TOTAL ← N
   SUMA ← 0
   mientras TOTAL > 0 hacer
      leer(NÚMERO)
      SUMA ← SUMA + NÚMERO
      TOTAL ← TOTAL - 1
   fin_mientras
   escribir('La suma de los', N, 'números es', SUMA)
fin
```

Cristian Pilapanta 1095211-2202102350

El bucle también podría terminarse si se pone cualquiera de estas condiciones:

- *Hasta_que* TOTAL sea cero
- *Desde* 1 *hasta* N

Para detener la ejecución de los bucles se utiliza una condición de parada. El pseudocódigo de una estructura repetitiva tendrá siempre este formato:

```
inicio
//inicialización de variables
repetir
   acciones S1, S2, ...
   salir según condición
   acciones Sn, Sn+1, ...
fin_repetir
```

Aunque la condición de salida se indica en el formato anterior en el interior del bucle (y existen lenguajes que así la contienen expresamente[1]), lo normal es que *la condición se indique al final o al principio del bucle*, y así se consideran tres tipos de instrucciones o estructuras repetitivas o iterativas generales, y una particular que denominaremos **iterar**, que contiene la salida en el interior del bucle.

```
iterar            (loop)
mientras          (while)
hacer-mientras    (do-while)
repetir           (repeat)
desde             (for)
```

El algoritmo de suma anterior podría expresarse en pseudocódigo estándar de la siguiente manera:

```
algoritmo SUMA_números
var
   entero : N, TOTAL
   real : NÚMERO, SUMA
inicio
   leer(N)
   TOTAL ← N
   SUMA ← 0
   repetir
      leer(NÚMERO)
      SUMA ← SUMA + NÚMERO
      TOTAL ← TOTAL - 1
   hasta_que TOTAL = 0
   escribir('La suma es', SUMA)
fin
```

Los tres casos generales de estructuras repetitivas dependen de la situación y modo de la condición, que se evalúa tan pronto se encuentra en el algoritmo, y su resultado producirá los tres tipos de estructuras citadas.

1. La condición de salida del bucle se realiza al principio del bucle (estructura **mientras**).

```
algoritmo SUMA1
inicio
```

[1] Modula-2 entre otros.

Cristian Pilapanta 1096211-2202102351

```
//Inicializar K, S a cero
   K ← 0
   S ← 0
leer(n)
mientras K < n hacer
      K ← K + 1
      S ← S + K
   fin_mientras
   escribir (S)
fin
```

Se ejecuta el bucle *mientras* se verifica una condición (`K < n`).

2. La condición de salida se origina al final del bucle; el cual se ejecuta *hasta que* se verifica una cierta condición.

```
repetir
   K ←K + 1
   S ←S + K
hasta_que K > n
```

3. La condición de salida se realiza con un contador que cuenta el número de iteraciones.

```
desde i = vi hasta vf hacer
S ← S + i
fin_desde
```

`i` es un contador que cuenta desde el valor inicial (`vi`) hasta el valor final (`vf`) con los incrementos que se consideren, si no se indica nada, el incremento es 1.

Cristian Palpanta 1095211-2202102352

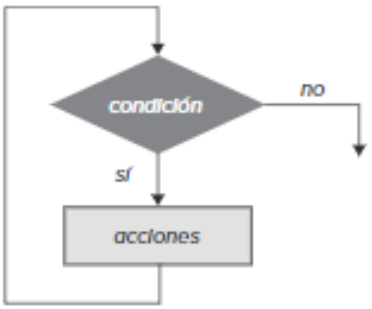
5.2. Estructura mientras ("while")

La estructura repetitiva **mientras** (en inglés **while** o **do-while**: *hacer mientras*) es aquella en la que el cuerpo del bucle se repite mientras se cumple una determinada condición. Cuando se ejecuta la instrucción **mientras**, la primera cosa que sucede es que se evalúa la condición (una expresión *booleana*). Si se evalúa *falsa*, no se toma ninguna acción y el programa prosigue en la siguiente instrucción del bucle. Si la expresión *booleana* es *verdadera*, entonces se ejecuta el cuerpo del bucle, después de lo cual se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez **mientras** la expresión *booleana* (condición) sea verdadera. El ejemplo anterior quedaría así y sus representaciones gráficas como las que se muestran en la figura 5.1.

EJEMPLO 5.1

Leer por teclado un número que represente una cantidad de números que a su vez se leerán también por teclado. Calcular la suma de todos esos números.

```
algoritmo suma_números
var
   entero : N, TOTAL
   real : número, SUMA
inicio
   leer(N)
   {leer número total N}
   TOTAL ←N
   SUMA ←0
```



a)

c)

Pseudocódigo en español

```
mientras condición hacer
   acción S1
   acción S2
   .
   .
   acción Sn
fin_mientras
```

Pseudocódigo en inglés

```
while condición do
   <acciones>
   .
   .
endwhile
```

o bien

```
dowhile condición
   <acciones>
   .
   .
enddo
```

b)

Figura 5.1. Estructura **mientras**: *a*) diagrama de flujo, *b*) pseudocódigo, *c*) diagrama N-S.

```
mientras TOTAL > 0 hacer
      leer(número)
      SUMA ← SUMA + número
      TOTAL ← TOTAL - 1
   fin_mientras
   escribir('La suma de los', N, 'números es', SUMA)
fin
```

En el caso anterior, como la variable TOTAL se va decrementando y su valor inicial era N, cuando tome el valor 0, significará que se han realizado N iteraciones, o, lo que es igual, se han sumado N números y el bucle se debe parar o terminar.

EJEMPLO 5.2

Contar los números enteros positivos introducidos por teclado. Se consideran dos variables enteras número *y* contador *(contará el número de enteros positivos). Se supone que se leen números positivos y se detiene el bucle cuando se lee un número negativo o cero.*

```
algoritmo cuenta_enteros
var
   entero : número, contador
inicio
   contador ← 0
   leer(número)
   mientras número > 0 hacer
      leer(número)
      contador ← contador + 1
   fin_mientras
   escribir('El número de enteros positivos es', contador)
fin
```

Cristian Palapanta 1065211-2202102353

```
inicio
contador ← 0
leer número
mientras número > 0
    leer número
    contador ← contador + 1
escribir 'números enteros', contador
fin
```

La secuencia de las acciones de este algoritmo se puede reflejar en el siguiente pseudocódigo:

Paso	Pseudocódigo	Significado
1	`contador ← 0`	Inicializar contador a 0
2	`leer(número)`	leer primer número
3	`mientras número > 0 hacer`	comprobar si `número > 0`. Si es así, continuar con el paso 4. Si no, continuar con el paso 7
4	`sumar 1 a contador`	incrementar `contador`
5	`leer(número)`	leer siguiente `número`
6	`regresar al paso 3`	evaluar y comprobar la expresión booleana
7	`escribir(contador)`	visualizar resultados

Observe que los pasos 3 a 6 se ejecutarán mientras los números de entrada sean positivos. Cuando se lea –15 (después de 4 pasos), la expresión `número > 0` produce un resultado falso y se transfiere el control a la acción **escribir** y el valor del contador será 4.

5.2.1. Ejecución de un bucle (lazo) cero veces

Observe que en una estructura **mientras** la primera cosa que sucede es la evaluación de la expresión booleana; si se evalúa *falsa* en ese punto, entonces el cuerpo del bucle nunca se ejecuta. Puede parecer *inútil* ejecutar el cuerpo del bucle *cero veces*, ya que no tendrá efecto en ningún valor o salida. Sin embargo, a veces es la acción deseada.

```
inicio
   n ← 5
   s ← 0
   mientras n <= 4 hacer
      leer(x)
      s ← s + x
  fin_mientras
fin
```

En el ejemplo anterior se aprecia que nunca se cumplirá la condición (expresión booleana `n < = 4`), por lo cual se ejecutará la acción **fin** y no se ejecutará ninguna acción del bucle.

Cristian Pilapanta 1095211-2202102354

EJEMPLO 5.3

El siguiente bucle no se ejecutará si el primer número leído es negativo o cero.

```
C ← 0
leer(número)
mientras número > 0 hacer
   C ← C + 1
   leer(número)
fin_mientras
```

5.2.2. Bucles infinitos

Algunos bucles no exigen fin y otros no encuentran el fin por error en su diseño. Por ejemplo, un sistema de reservaciones de líneas aéreas puede repetir un bucle que permita al usuario añadir o borrar reservaciones. El programa y el bucle corren siempre, o al menos hasta que la computadora se apaga. En otras ocasiones un bucle no se termina nunca, porque nunca se cumple la condición.

Un bucle que nunca se termina se denomina *bucle infinito* o *sin fin*. Los bucles sin fin no intencionados son perjudiciales para la programación y se deben evitar siempre.

Consideremos el siguiente bucle que visualiza el interés producido por un capital a las tasas de interés comprendidas en el rango desde 10 a 20%.

```
leer(capital)
tasa ← 10
mientras tasa <> 20 hacer
   interés ← tasa*0.01*capital // tasa*capital/100=tasa*0.01*capital
   escribir('interés producido', interés)
   tasa ← tasa + 2
fin_mientras
escribir('continuación')
```

Los sucesivos valores de la tasa serán 10, 12, 14, 16, 18, 20, de modo que al tomar *tasa*, el valor 20 detendrá el bucle y se escribirá el mensaje 'continuación'. Supongamos que se cambia la última línea del bucle por

```
tasa ← tasa + 3
```

El problema es que el valor de la tasa salta ahora de 19 a 22 y nunca será igual a 20 (10, 13, 16, 19, 22,...). El bucle sería infinito, la expresión booleana para terminar el bucle será:

`tasa < 20` *o bien* `tasa <= 20`

Regla práctica

Las pruebas o test en las expresiones booleanas es conveniente que sean *mayor* o *menor que* en lugar de pruebas de *igualdad* o *desigualdad*. En el caso de la codificación en un lenguaje de programación, esta regla debe seguirse rigidamente en el caso de comparación de *números reales*, ya que como esos valores se almacenan en cantidades aproximadas, las comparaciones de igualdad de valores reales normalmente plantean problemas. Siempre que realice comparaciones de números reales use las relaciones <, <=, > o >=.

5.2.3. Terminación de bucles con datos de entrada

Si su algoritmo o programa está leyendo una lista de valores con un bucle **mientras**, se debe incluir algún tipo de mecanismo para terminar el bucle. Existen cuatro métodos típicos para terminar un bucle de entrada:

1. Preguntar antes de la iteración.
2. Encabezar la lista de datos con su tamaño.

Cristian Pilapana 1095211-2202102355

3. Finalizar la lista con su valor de entrada.
4. Agotar los datos de entrada.

Examinémoslos por orden. El primer método simplemente solicita con un mensaje al usuario si existen más entradas.

```
Suma ← 0
escribir('Existen más números en la lista s/n')
leer(Resp) //variable Resp, tipo
carácter mientras(Resp = 'S') o (Resp = 's')
hacer
   escribir('número')
   leer(N)
   Suma ← Suma + N
   escribir('Existen más números (s/n)')
   leer(Resp)
fin_mientras
```

En ocasiones, este método es aceptable y muy útil, pero suele ser tedioso para listas grandes; en este caso, es preferible incluir una señal de parada. El método de conocer en la cabecera del bucle el tamaño o el número de iteraciones ya ha sido visto en ejemplos anteriores.

Probablemente el mejor método para terminar un bucle que lee una lista de valores es con un *valor centinela*, que es un valor especial usado para indicar el final de una lista de datos. Por ejemplo, supongamos que se tienen las calificaciones de unos exámenes (cada calificación comprendida entre 0 y 100); un valor centinela en esta lista puede ser –999, ya que nunca será una calificación válida y cuando aparezca este valor se terminará el bucle. Si la lista de datos son números positivos, un valor centinela puede ser un número negativo que indique el final de la lista. El siguiente ejemplo realiza la suma de todos los números positivos introducidos desde la terminal.

```
suma ← 0
leer(número)
mientras número >= 0 hacer
   suma ← suma + número
   leer(número)
fin_mientras
```

Observe que el último número leído de la lista no se añade a la suma si es negativo, ya que se sale fuera del bucle. Si se desea sumar los números 1, 2, 3, 4 y 5 con el bucle anterior, el usuario debe introducir, por ejemplo:

```
1 2 3 4 5 -1
```

El valor final –1 se lee, pero no se añade a la suma. Observe que cuando se usa un valor centinela se invierte el orden de las instrucciones de lectura y suma con un valor centinela, y éste debe leerse al final del bucle y debe tener la instrucción `leer` al final del mismo.

El último método de agotamiento de datos de entrada es comprobar simplemente que no existen más datos de entrada. Este sistema suele depender del tipo de lenguaje, por ejemplo, Pascal puede detectar el final de una línea; en los archivos secuenciales se puede detectar el final físico del archivo (*eof*, `end of file`).

EJEMPLO 5.4

Considere los siguientes algoritmos. ¿Qué visualizará y cuántas veces se ejecuta el bucle?

1.
```
i ← 0
mientras i < 6 hacer
   escribir(i)
   i ← i + 1
fin_mientras
```

Cristian Pilapanta 1006211-2202102356

La salida es el valor de la variable de control *i* al principio de cada ejecución del cuerpo del bucle: 0, 1, 2, 3, 4 y 5. El bucle se ejecuta seis veces.

2.
```
i ← 0
mientras i < 6 hacer
   i ← i + 1
   escribir(i)
fin_mientras
```

La salida será entonces 1, 2, 3, 4, 5 y 6; el cuerpo del bucle se ejecuta también seis veces. Observe que cuando `i` = 5, la expresión *booleana* es verdadera y el cuerpo del bucle se ejecuta; con `i` = 6 la sentencia `escribir` se ejecuta, pero a continuación se evalúa la expresión *booleana* y se termina el bucle.

EJEMPLO 5.5

Calcular la media de un conjunto de notas de alumnos. Pondremos un valor centinela de –99 que detecte el fin del bucle.

```
inicio
   total ← 0
   n ← 0     //número de alumnos
   leer(nota) //la primera nota debe ser distinta de -99
   mientras nota <> -99 hacer
      total ← total + nota
      n ← n + 1
      leer (nota)
   fin_mientras
   media ← total / n
   escribir('La media es', media)
fin
```

Observe que `total` y `n` se inicializan a cero antes de la instrucción **`mientras`**. Cuando el bucle termina, la variable `total` contiene la suma de todas las notas y, por consiguiente, `total/n`, siendo `n` el número de alumnos, será la media de la clase.

5.3. Estructura hacer-mientras ("do-while")

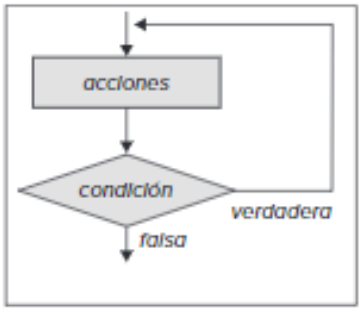
Tanto el bucle `mientras` como `desde` evalúan la expresión al comienzo del bucle de repetición; siempre se utilizan para crear un bucle *pre-test*, también denominados bucles controlados por la entrada. En numerosas ocasiones se necesita que el conjunto de sentencias que componen el cuerpo del bucle se ejecuten al menos una vez, sea cual sea el valor de la expresión o condición de evaluación. Estos bucles se denominan bucles *post-test* o bucles controlados por la salida. Un caso típico es el bucle **`hacer-mientras (do-while)`** existente en lenguajes como **C/C++**, **Java** o **C♯**.

El bucle **`hacer-mientras`** es análogo al bucle **`mientras`**, y el cuerpo del bucle se ejecuta una y otra vez mientras la condición (expresión *booleana*) sea verdadera. Existe, sin embargo, una gran diferencia y es que el cuerpo del bucle está encerrado entre las palabras reservadas `hacer` y `mientras`, de modo que las sentencias de dicho cuerpo se ejecutan, al menos una vez, antes de que se evalúe la expresión booleana. En otras palabras, el cuerpo del bucle siempre se ejecuta, al menos una vez, incluso aunque la expresión *booleana* sea falsa.

Regla

El bucle **`hacer-mientras`** se termina de ejecutar cuando el valor de la condición es falsa. La elección entre un bucle **`mientras`** y uno **`hacer-mientras`** depende del problema de cómputo a resolver. En la mayoría de los casos, la condición de entrada del bucle `mientras` es la elección correcta. Por ejemplo, si el bucle se utiliza para recorrer una lista de números (o una lista de cualquier tipo de objetos), la lista puede estar vacía, en cuyo caso las sentencias del bucle nunca se ejecutarán. Si se aplica un bucle **`hacer-mientras`**, nos conduce a un código de errores.

Cristian Pilapanta 1006211-2202102356



a) Diagrama de flujo de una sentencia **hacer-mientras**

```
hacer
   <acciones>
mientras (<expresión>)
```

b) Pseudocódigo de una sentencia **hacer-mientras**

Figura 5.2. Estructura **hacer-mientras**: *a*) diagrama de flujo, *b*) pseudocódigo.

Al igual que en el caso del bucle **mientras**, la sentencia en el interior del bucle puede ser simple o compuesta. Todas las sentencias en el interior del bucle se ejecutan al menos una vez antes de que la expresión o condición se evalúe. Entonces, si la expresión es **verdadera** (un valor distinto de cero, en C/C++), las sentencias del cuerpo del bucle se ejecutan una vez más. El proceso continúa hasta que la expresión evaluada toma el valor **falso** (valor cero en C/C++). El diagrama de control del flujo se ilustra en la figura 5.2, donde se muestra el funcionamiento de la sentencia **hacer-mientras**. La figura 5.3 representa un diagrama de sintaxis con notación BNF de la sentencia **hacer-mientras**.

Cristian Pilapanta 1095211-2202102357

Sentencia hacer-mientras::=

```
hacer
   <cuerpo del bucle>
mientras (<condición_del_bucle>)
```

donde

```
<cuerpo del bucle>::= <sentencia>
                  ::= <sentencia_compuesta>

<condición del bucle>::= <expresión booleana>
```

Nota: el cuerpo del bucle se repite mientras <*condición del bucle*> sea verdadero.

Figura 5.3. Diagrama de sintaxis de la sentencia **hacer-mientras**.

EJEMPLO 5.6

Obtener un algoritmo que lea un número (por ejemplo, 198) y obtenga el número inverso (por ejemplo, 891).

```
algoritmo invertir número
var
   entero: núm, dígitoSig
inicio
   núm ← 198
   escribir ('Número: ← ', núm)
```

```
   escribir ('Número en orden inverso:')
   hacer
      dígitoSig ← núm MOD 10
      escribir(dígitoSig)
      núm ← núm DIV 10
   mientras núm > 0
fin
```

La salida de este programa se muestra a continuación:

```
Número:  198
Número en orden inverso:  891
```

Análisis del ejemplo anterior

Con cada iteración se obtiene el dígito más a la derecha, ya que es el resto de la división entera del valor del número (`núm`) por 10. Así, en la primera iteración `dígitoSig` vale 8, ya que es el resto de la división entera de 198 entre 10 (cociente 19 y resto 8). Se visualiza el valor 8. A continuación se divide 198 entre 10 y se toma el cociente entero 19, que se asigna a la variable `núm`.

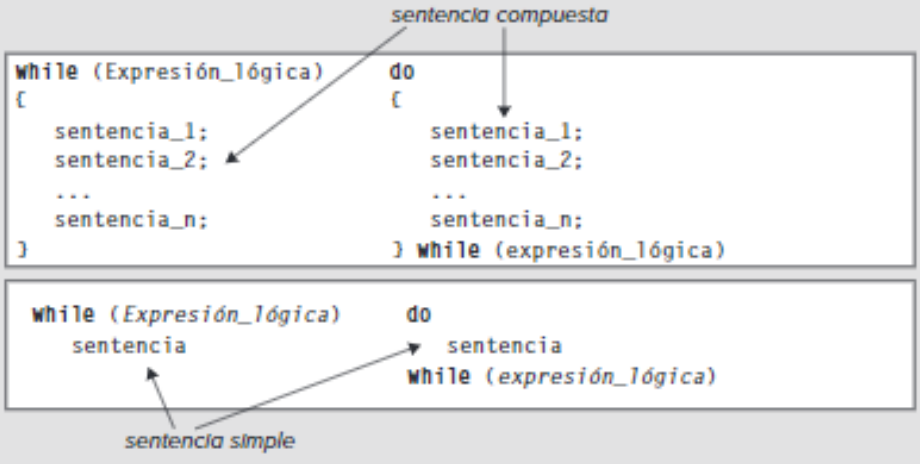
En la siguiente iteración se divide 19 entre 10 (cociente entero 1, resto 9) y se visualiza, por consiguiente, el valor del resto, `dígitoSig`, es decir, el dígito 9; a continuación se divide 19 entre 10 y se toma el cociente entero, es decir, 1. En la tercera y última iteración se divide 1 entre 10, y se toma el resto (`dígitoSig`) que es el dígito 1. Se visualiza el dígito 1 a continuación de 89, y como resultado final aparece 891. Luego se efectúa la división de nuevo entre 10 y entonces el cociente entero es 0 que se asigna a `núm`, que al no ser ya mayor que cero hace que se termine el bucle y el algoritmo correspondiente.

Cristian Pilapanta 1006211-2202110019

5.4. Diferencias entre `mientras (while)` y `hacer-mientras (do-while)`: una aplicación en C++

Una sentencia **`do-while`** es similar a una sentencia **`while`**, excepto que el cuerpo del bucle se ejecuta siempre al menos una vez.

Sintaxis



EJEMPLO 1

```
// cuenta a 10
int x = 0;
do
   cout << "X:" << x++;
while (x < 10)
```

EJEMPLO 2

```
// imprimir letras minúsculas del alfabeto
char car = 'a';
do
{
   cout << car << '';
   car++;
}while (car <= 'z');
```

EJEMPLO 5.7

Visualizar las potencias de dos cuerpos cuyos valores estén en el rango 1 a 1000.

```
// ejercicio con while
potencia = 1;
while (potencia < 1000)
{
   cout << potencia << endl;
   potencia *= 2
} // fin de while
```

```
// ejercicio con do-while
potencia = 1;
do
{
   cout << potencia << endl;
      potencia *= 2;
} while (potencia < 1000);
```

Cristian Pilapanta 1095211-2202110220

5.5. Estructura repetir ("repeat")

Existen muchas situaciones en las que se desea que un bucle se ejecute al menos una vez *antes* de comprobar la condición de repetición. En la estructura **mientras** si el valor de la expresión booleana es inicialmente falso, el cuerpo del bucle no se ejecutará; por ello, se necesitan otros tipos de estructuras repetitivas.

La estructura **repetir (repeat)** se ejecuta hasta que se cumpla una condición determinada que se comprueba al final del bucle (figura 5.4).

El bucle **repetir-hasta_que** se repite mientras el valor de la expresión *booleana* de la condición sea *falso*, justo la opuesta de la sentencia **mientras**.

```
algoritmo repetir
var
   real : número
   entero: contador
inicio
   contador ← 1
   repetir
      leer(número)
      contador ← contador + 1
   hasta_que contador > 30
   escribir('Números leídos 30')
fin
```

En el ejemplo anterior el bucle se repite hasta que el valor de la variable `contador` exceda a 30, lo que sucederá después de 30 ejecuciones del cuerpo del bucle.

Pseudocódigo

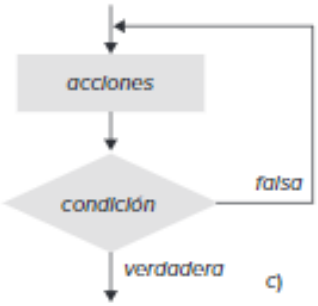
```
repetir
   <acciones>
   .
   .
hasta_que <condición>
```

a) Español

```
repeat
   <acciones>
   .
   .
until <condición>
```

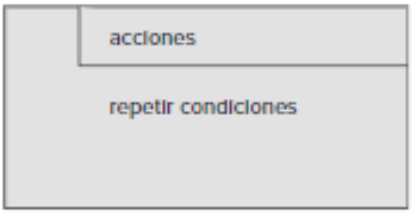
b) Inglés

Diagrama de flujo



c)

Diagrama N-S



b)

Figura 5.4. Estructura **repetir**: pseudocódigo, diagrama de flujo, diagrama N-S.

EJEMPLO 5.8

Desarrollar el algoritmo necesario para calcular el factorial de un número N que responda a la fórmula:

$N! = N * (N - 1) * (N - 2), ..., 3 * 2 * 1$

El algoritmo correspondiente es:

```
algoritmo factorial
var
   entero : I, N
   real : Factorial
inicio
   leer(N)          // N > = 1
   Factorial ← 1
   I ← 1
   repetir
      Factorial ← Factorial * I
      I ← I + 1
   hasta_que I = N + 1
   escribir('El factorial del número', N, 'es', Factorial)
fin
```

Con una estructura **repetir** el cuerpo del bucle *se ejecuta siempre al menos una vez*. Cuando una instrucción **repetir** se ejecuta, lo primero que sucede es la ejecución del bucle y, a continuación, se evalúa la expresión *booleana* resultante de la condición. Si se evalúa como falsa, el cuerpo del bucle se repite y la expresión *booleana* se evalúa una vez. Después de cada iteración del cuerpo del bucle, la expresión *booleana* se evalúa; si es *verdadera*, el bucle termina y el programa sigue en la próxima instrucción a **hasta_que**.

Cristian Pilapanta 1006211-2202110220

Diferencias de las estructuras mientras y repetir

- La estructura **mientras** termina cuando la condición es falsa, mientras que **repetir** termina cuando la condición es verdadera.
- En la estructura **repetir** el cuerpo del bucle se ejecuta siempre al menos una vez; por el contrario, **mientras** es más general permite la posibilidad de que el bucle pueda no ser ejecutado. Para usar la estructura **repetir** debe estar seguro de que el cuerpo del bucle (bajo cualquier circunstancia) se repetirá al menos una vez.

EJEMPLO 5.9

Encontrar el entero positivo más pequeño (núm) para el cual la suma 1+2+3+...+núm es menor o igual que límite.

1. Introducir `límite`.
2. Inicializar `núm` y `suma` a 0.
3. Repetir las acciones siguientes hasta que `suma > límite`:
   - Incrementar `núm` en 1.
   - Añadir `núm` a `suma`.
4. Visualizar `núm` y `suma`.

El pseudocódigo de este algoritmo es:

```
algoritmo más_pequeño
var
   entero : núm, límite, suma
inicio
   leer(límite)
   núm ← 0
   suma ← 0
   repetir
      núm ← núm + 1
      suma ← suma + núm
   hasta_que suma > límite
   escribir(núm, suma)
fin
```

EJEMPLO 5.10

Escribir los números 1 a 100.

```
algoritmo uno_cien
var
   entero : núm
inicio
   núm ← 1
   repetir
      escribir(núm)
      núm ← núm + 1
   hasta_que núm > 100
fin
```

Cristian Pilapanta 1095211-2202110021

EJEMPLO 5.11

Es muy frecuente tener que realizar validación de entrada de datos en la mayoría de las aplicaciones. Este ejemplo detecta cualquier entrada comprendida entre 1 y 12, rechazando las restantes, ya que se trata de leer los números correspondientes a los meses del año.

```
algoritmo validar_mes
var
   entero : mes
inicio
   escribir('Introducir número de mes')
   repetir
      leer(mes)
      si (mes < 1) o (mes > 12) entonces
         escribir('Valor entre 1 y 12')
      fin_si
   hasta_que (mes >= 1) y (mes <= 12)
fin
```

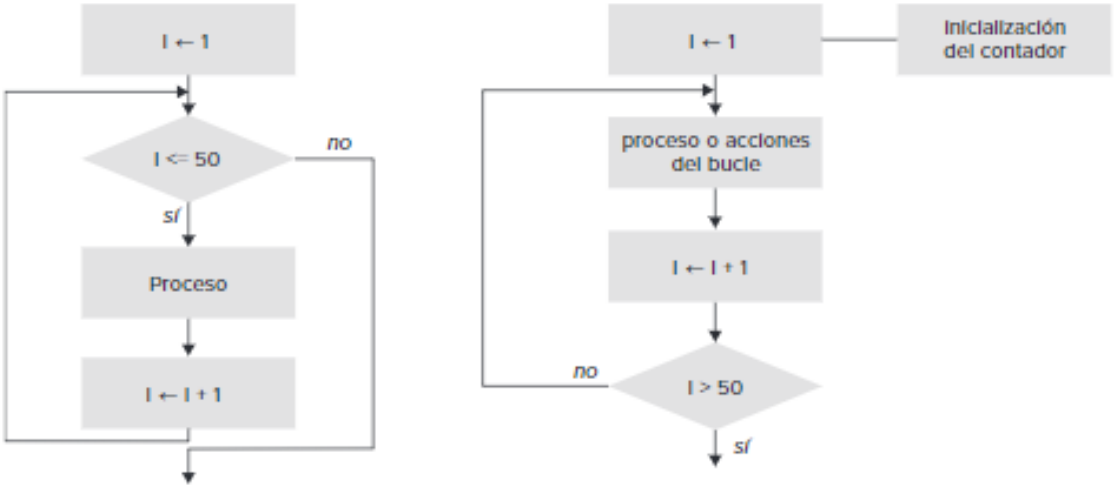
Este sistema es conocido como *interactivo* por entablar un "diálogo imaginario" entre la computadora y el programador que se produce "en tiempo real" entre ambas partes, es decir, "interactivo" con el usuario.

5.6. Estructura desde/para ("for")

En muchas ocasiones se conoce de antemano el número de veces que se desea ejecutar las acciones de un bucle. En estos casos, en el que el número de iteraciones es fijo, se debe usar la estructura **desde** o **para** (**for**, en inglés). La estructura **desde** ejecuta las acciones del cuerpo del bucle un número especificado de veces, y de modo automático controla el número de iteraciones o pasos a través del cuerpo del bucle. Las herramientas de programación de la estructura **desde** o **para** se muestran en la página siguiente junto a la figura 5.5.

5.6.1. Otras representaciones de estructuras repetitivas desde/para (for)

Un bucle **desde (for)** se representa con los símbolos de proceso y de decisión mediante un contador. Así, por ejemplo, en el caso de un bucle de lectura de cincuenta números para tratar de calcular su suma:



Cristian Pilapanta 1095211-2202110022

Pseudocódigo estructura **desde**

```
desde v ← vi hasta vf [incremento incr] hacer
<acciones>
    .
    .
    .
fin_desde
v: variable índice
vi, vf: valores inicial y final de la variable
```

a) Modelo 1

```
para v ← vi hasta vf [incremento incr] hacer
    <acciones>
    .
    .
    .
fin_para
```

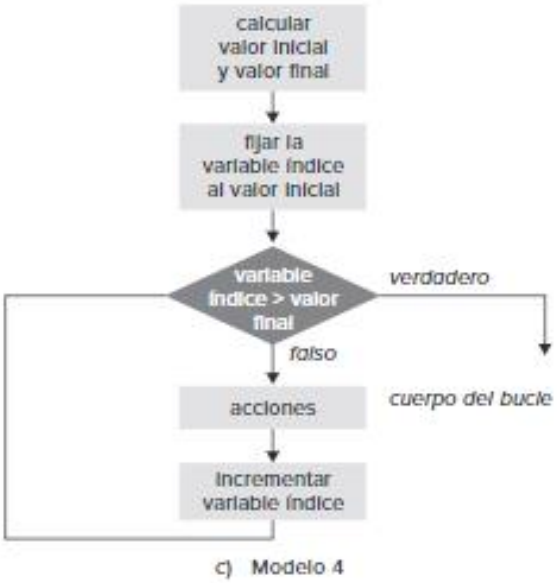
b) Modelo 2

Diagrama N-S, estructura **desde**

```
desde v ← vi hasta vf [incremento incr] hacer
                    <acciones>
fin_desde
```

b) Modelo 3

Diagrama de flujo, estructura, **desde**

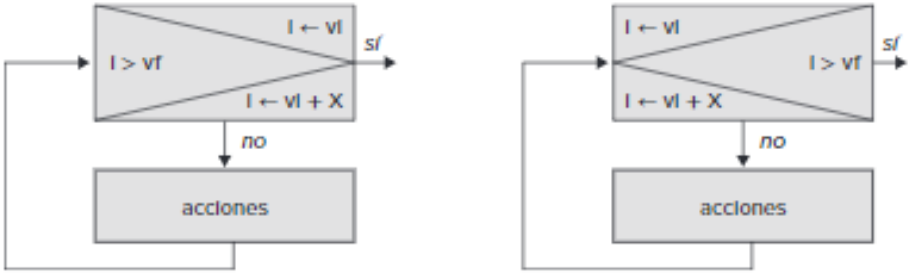


c) Modelo 4

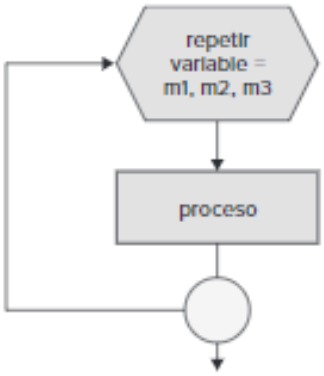
Figura 5.5. Estructura **desde (for)**: *a)* pseudocódigo, *b)* diagrama N-S, *c)* diagrama de flujo.

Cristian Pilaparita 1095211-2202110023

Es posible representar el bucle con símbolos propios:

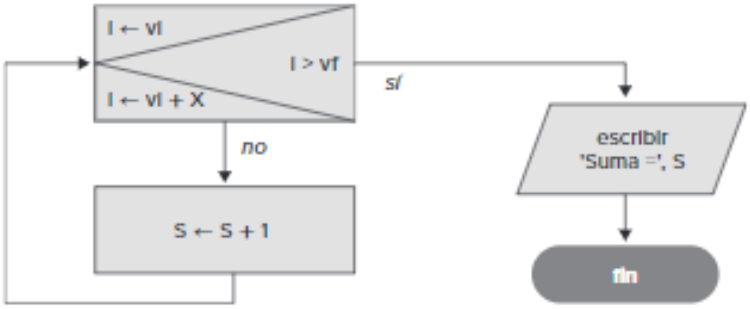


o bien mediante este otro símbolo:



m1 = *contador inicial*
m2 = *contador final*
m3 = *incremento de paso*

Como aplicación, *calcular la suma de los N primeros enteros.*



Equivale a:

```
algoritmo suma
var
   entero : T, N, S
inicio
   S ← 0
   desde I ← 1 hasta N hacer
      S ← S + I
   fin_desde
   escribir('Suma =', S)
fin
```

Cristian Pallarola 1066211-2202110024

La estructura **desde** comienza con un valor inicial de la variable índice y las acciones especificadas se ejecutan, a menos que el valor inicial sea mayor que el valor final. La variable índice se incrementa en uno y si este nuevo valor no excede al final, se ejecutan de nuevo las acciones. Por consiguiente, las acciones específicas en el bucle se ejecutan para cada valor de la variable índice desde el valor inicial hasta el valor final con el incremento de uno en uno.

El incremento de la variable índice siempre es 1 si no se indica expresamente lo contrario. Dependiendo del tipo de lenguaje, es posible que el incremento sea distinto de uno, positivo o negativo. Así, por ejemplo, Fortran admite diferentes valores positivos o negativos del incremento, y Pascal sólo admite incrementos cuyo tamaño es la unidad: bien positivos bien negativos. La variable índice o de control normalmente será de tipo entero, y es normal emplear como nombres las letras `I`, `J`, `K`.

El formato de la estructura **desde** varía si se desea un incremento distinto a 1, bien positivo, bien negativo (decremento).

```
desde v ← vi hasta vf          inc paso hacer          {inc, incremento}
                               dec                     {dec, decremento}
   <acciones>
   .
   .
   .
fin_desde
```

Si el valor inicial de la variable índice es menor que el valor final, los incrementos deben ser positivos, ya que en caso contrario la secuencia de acciones no se ejecutaría. De igual modo, si el valor inicial es mayor que el valor final, el incremento debe ser negativo, es decir, *decremento*. Al incremento se le suele denominar también *paso* (*step*, en inglés). Es decir:

```
desde i ← 20 hasta 10 hacer
   <acciones>
fin_desde
```

no se ejecutaría, ya que el valor inicial es 20 y el valor final 10, y como se supone un incremento positivo, de valor 1, se produciría un error. El pseudocódigo correcto debería ser:

```
desde i ← 20 hasta 10 decremento 1 hacer
   <acciones>
fin_desde
```

5.6.2. Realización de una estructura desde con estructura mientras

Es posible, como ya se ha mencionado, sustituir una estructura **desde** por una **mientras**; en las líneas siguientes se indican dos formas para ello:

1. *Estructura* **desde** *con incrementos de la variable índice positivos.*

```
v ← vi
mientras v <= vf hacer
   <acciones>
   v ← v + incremento
fin_mientras
```

2. *Estructura* **desde** *con incrementos de la variable índice negativos.*

```
v ← vi
mientras v >= vf hacer
```

Cristian Pilapanta 1095211-2202110025

```
    <acciones>
    v ← v - decremento
fin_mientras
```

La estructura **desde** puede realizarse con algoritmos basados en estructura **mientras** y **repetir**, por lo que pueden ser intercambiables cuando así lo desee. Las estructuras equivalentes a **desde** son las siguientes:

a)
```
inicio
    i ← n
    mientras i > 0 hacer
        <acciones>
        i ← i - 1
    fin_mientras
fin
```

b)
```
inicio
    i ← 1
    mientras i <= n hacer
        <acciones>
        i ← i + 1
    fin_mientras
fin
```

c)
```
inicio
    i ← 0
    repetir
        <acciones>
        i ← i+1
    hasta_que i = n
fin
```

d)
```
inicio
    i ← 1
    repetir
        <acciones>
        i ← i+1
    hasta_que i > n
fin
```

e)
```
inicio
    i ← n + 1
    repetir
        <acciones>
        i ← i - 1
    hasta_que i = 1
fin
```

f)
```
inicio
    i ← n
    repetir
        <acciones>
        i ← i - 1
    hasta_que i < 1
fin
```

Cristian Pilapanta 1095211-2202110026

5.7. Salidas internas de los bucles

Aunque no se incluye dentro de las estructuras básicas de la programación estructurada, en ocasiones es necesario disponer de una estructura repetitiva que permita la salida en un punto intermedio del bucle cuando se cumpla una condición. Esta nueva estructura sólo está disponible en algunos lenguajes de programación específicos; la denominaremos **iterar** para diferenciarlo de **repetir_hasta** ya conocida. Las salidas de bucles suelen ser válidas en estructuras **mientras**, **repetir** y **desde**.

El formato de la estructura es:

```
iterar
    <acciones>
    si <condición> entonces
        salir_bucle
    fin_si
    <acciones>
fin_iterar
```

En general, la instrucción **iterar** no produce un programa legible y comprensible como lo hacen **mientras** y **repetir**. La razón de esta ausencia de claridad es que la salida de un bucle ocurre en medio de éste, mientras que normalmente la salida del bucle es al principio o al final del mismo. Le recomendamos no recurra a esta opción (aunque la tenga su lenguaje), a menos que no exista otra alternativa o disponga de la estructura **iterar** (*loop*).

EJEMPLO 5.12

Una aplicación de un posible uso de la instrucción `salir` *se puede dar cuando se incluyen mensajes de petición en el algoritmo para la introducción sucesiva de informaciones.*

Algoritmo 1

```
leer(información)
repetir
   procesar (información)
   leer(información)
hasta_que fin_de_lectura
```

Algoritmo 2

```
leer(información)
mientras_no fin_de_lectura
   procesar (información)
   leer(información)
fin_mientras
```

En los algoritmos anteriores cada entrada (lectura) de información va acompañada de su correspondiente proceso, pero la primera lectura está fuera del bucle. Se pueden incluir en el interior de éste todas las lecturas de información si se posee una estructura **salir (exit)**. Un ejemplo de ello es la estructura siguiente:

```
iterar
   leer(información)
   si fin_de_lectura entonces
      salir_bucle
   fin_si
   procesar (información)
fin_iterar
```

Cristian Pilapanta 1006211-2202110027

5.8. Sentencias de salto interrumpir (break) y continuar (continue)

Las secciones siguientes examinan las sentencias de salto (*jump*) que se utilizan para influir en el flujo de ejecución durante la ejecución de una sentencia de bucle. Las sentencias más utilizadas son: **interrumpir (break)**, **continuar (continue)** a las que es preciso añadir, la ya mencionada, **goto (ir_a)** y **volver** o **retornar (return)** para la terminación inmediata de funciones o métodos.

5.8.1. Sentencia interrumpir (break)

En ocasiones, los programadores desean terminar un bucle en un lugar determinado del cuerpo de éste, en vez de esperar a que termine de modo natural por su entrada o por su salida. Un método de conseguir esta acción (siempre utilizado con precaución y con un control completo del bucle) es mediante la sentencia `interrumpir (break)` que suele utilizarse en la sentencia `según_sea (switch)`.

La sentencia `interrumpir` puede utilizarse para terminar una sentencia de iteración, y cuando se ejecuta, produce que el flujo de control salte fuera a la siguiente sentencia inmediatamente después de la de iteración. La sentencia `interrumpir` se puede colocar en el interior del cuerpo del bucle para implementar este efecto.

Sintaxis

```
interrumpir
sentencia_interrumpir::= interrumpir
```

EJEMPLO 5.13

```
hacer
   escribir ('Introduzca un número de identificación')
   leer (númId)
   si (númId < 1000 o númId > 1999)entonces
      escribir ('Número no válido')
      escribir ('Por favor, introduzca otro número')
   si-no
      interrumpir
   fin_si
mientras (expresión cuyo valor sea siempre verdadero)
```

EJEMPLO 5.14

```
var entero: t
desde t ← 0 hasta t < 100 incremento 1 hacer
   escribir (t)
   si (t = ld) entonces
      interrumpir
   fin_si
fin_desde
```

Regla

La sentencia **interrumpir (break)** se utiliza con frecuencia junto con una sentencia si **(if)**, actuando como una condición interna del bucle.

5.8.2. Sentencia continuar (continue)

La sentencia **continuar (continue)** hace que el flujo de ejecución salte el resto de un cuerpo del bucle para continuar con el siguiente bucle o iteración. Esta característica suele ser útil en algunas circunstancias.

Sintaxis

```
continuar
Sentencia_continuar:= continuar
```

La sentencia **continuar** sólo puede utilizarse dentro de una *iteración de un bucle;* esta sentencia no interfiere con el número de veces que se repite el cuerpo del bucle como sucede con interrumpir, sino que simplemente influye en el flujo de control en cualquier iteración específica.

EJEMPLO 5.15

```
i = 0
desde i = 0 hasta 20 inc 1 hacer
   si (i mod 4 = 0) entonces
      continuar
   fin_si
   escribir (i, ', ')
fin_desde
```

Cristian Palapanta 1095211-2202110028

Al ejecutar el bucle anterior se producen estos resultados:

```
1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15, 17, 18, 19
```

Un análisis del algoritmo nos proporciona la razón de los resultados anteriores:

1. La variable `i` se declara igual a cero, como valor inicial.
2. El bucle `i` se incrementa en cada iteración en 1 hasta llegar a 21, momento en que se termina la ejecución del bucle.
3. Siempre que `i` es múltiplo de 4 (`i mod 4`) se ejecuta la sentencia **`continuar`** y salta el flujo del programa sobre el resto del cuerpo del bucle, se termina la iteración en curso y comienza una nueva iteración (en ese caso no se escribe el valor de `i`). En consecuencia, no se visualiza el valor de `i` correspondiente (múltiplo de 4).
4. Como resultado final se visualizan todos los números comprendidos entre 0 y 20, excepto los múltiplos de 4; es decir, 4, 8, 12, 16 y 20.

5.9. Comparación de bucles `while`, `for` y `do-while`: una aplicación en C++

C++ proporciona tres sentencias para el control de bucles: **`while`**, **`for`** y **`do-while`**. El bucle **`while`** se repite *mientras* su condición de repetición es verdadera; el bucle **`for`** se utiliza normalmente cuando el conteo esté implicado, o bien en donde el número de iteraciones requeridas se puede determinar al principio de la ejecución del bucle, o simplemente cuando existe una necesidad de seguir el número de veces que un suceso particular tiene lugar. El bucle **`do-while`** se ejecuta de un modo similar a **`while`** excepto que las sentencias del cuerpo del bucle se ejecutan siempre al menos una vez.

La tabla 5.1 describe cuándo se usa cada uno de los tres bucles. En C++, el bucle **`for`** es el más utilizado de los tres. Es relativamente fácil reescribir un bucle **`do-while`** como un bucle **`while`**, insertando una asignación inicial de la variable condicional. Sin embargo, no todos los bucles **`while`** se pueden expresar de modo adecuado como **`do-while`**, ya que un **`do-while`** se ejecutará siempre al menos una vez, y el **`while`** puede no ejecutarse. Por esta razón, un **`while`** suele preferirse a un bucle **`do-while`**, a menos que esté claro que se debe ejecutar una iteración como mínimo.

Tabla 5.1. Formatos de los bucles en C++

`while`	El uso más frecuente es cuando la repetición no está controlada por un contador; el test de condición precede a cada repetición; el cuerpo del bucle puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es falsa.
`for`	Bucle de conteo cuando el número de repeticiones se conoce por anticipado y puede ser controlado por un contador; también es adecuado para bucles que implican un control no contable del bucle con simples etapas de inicialización y de actualización; el test de la condición precede a la ejecución del cuerpo del bucle.
`do-while`	Es adecuada cuando se debe asegurar que se ejecuta el bucle al menos una vez.

Comparación de tres bucles

```
cuenta = valor_inicial;
while (cuenta < valor_parada)
{
   ...
   cuenta++;
} // fin de while
```

Cristian Pilapanta 1095211-2202110029

```
for(cuenta=valor_inicial; cuenta<valor_parada; cuenta++)
{
   ...
} // fin de for
cuenta = valor_inicial;
if (valor_inicial < valor_parada)
   do
   {
      ...
      cuenta++;
   } while (cuenta < valor_parada);
```

5.10. Diseño de bucles (lazos)

El diseño de un bucle requiere tres partes:

1. El cuerpo.
2. Las sentencias de inicialización.
3. Las condiciones para la terminación.

Cristian Pilapanta 1096211-2202110229

5.10.1. Bucles para diseño de sumas y productos

Muchas tareas frecuentes implican la lectura de una lista de números y calculan su suma. Si se conoce cuántos números habrá, tal tarea se puede ejecutar fácilmente por el siguiente pseudocódigo. El valor de la variable `total` es la cantidad de números que se suman. La suma se acumula en la variable `suma`.

```
suma ← 0;
repetir lo siguiente total veces:
   cin >> siguiente;
   suma ← suma + siguiente;
fin_bucle
```

Este código se implementa fácilmente con un bucle `for` en C++.

```
int suma = 0;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
   cin >> siguiente;
   suma = suma + siguiente;
}
```

Observe que la variable `suma` se espera tome un valor cuando se ejecuta la siguiente sentencia:

```
suma = suma + siguiente;
```

Dado que `suma` debe tener un valor la primera vez que la sentencia se ejecuta, `suma` debe estar inicializada a algún valor antes de que se ejecute el bucle. Con el objeto de determinar el valor correcto de inicialización de `suma` se debe pensar sobre qué sucede después de una iteración del bucle. Después de añadir el primer número, el valor de `suma` debe ser ese número. Esto es, la primera vez que se ejecute el bucle, el valor de `suma + siguiente` sea igual a `siguiente`. Para hacer esta operación *true* (verdadero), el valor de `suma` debe ser inicializado a 0.
Si en lugar de `suma`, se desea realizar productos de una lista de números, la técnica a utilizar es:

```
int producto = 1;
for (int cuenta = 1; cuenta <= total; cuenta++)
```

```
{
   cin >> siguiente;
   producto = producto * siguiente;
}
```

La variable `producto` debe tener un valor inicial. No se debe suponer que todas las variables se deben inicializar a cero. Si `producto` se inicializara a cero, seguiría siendo cero después de que el bucle anterior se terminara.

5.10.2. Fin de un bucle

Existen cuatro métodos utilizados normalmente para terminar un bucle de entrada y son:[2]

1. *Lista encabezada por tamaño.*
2. *Preguntar antes de la iteración.*
3. *Lista terminada con un valor centinela.*
4. *Agotamiento de la entrada.*

Lista encabezada por el tamaño

Si su programa puede determinar el tamaño de una lista de entrada por anticipado, bien preguntando al usuario o por algún otro método, se puede utilizar un bucle "`repetir n veces`" para leer la entrada exactamente `n` veces, en donde `n` es el tamaño de la lista.

Cristian Pilapanta 1095211-2202110030

Preguntar antes de la iteración

El segundo método para la terminación de un bucle de entrada es preguntar, simplemente, al usuario, después de cada iteración del bucle, si el bucle debe ser o no iterado de nuevo. Por ejemplo:

```
suma = 0;
cout << "¿Existen números en la lista?:\n"
     << "teclee S para Sí, N para No e Intro para Final):";
char resp;
cin >> resp;
while ((resp == 'S')|| (resp == 's'))
{
   cout << "Introduzca un número:";
   cin >> número;
   suma = suma + número;
   cout << "¿Existen más números?:\n";
        << "S para Sí, N para No. Final con Intro:";
   cin >> resp;
}
```

Este método es muy tedioso para listas grandes. Cuando se lea una lista larga es preferible incluir una única señal de parada, como se incluye en el método siguiente.

Valor centinela

El método más práctico y eficiente para terminar un bucle que lee una lista de valores del teclado es mediante un **valor centinela**, que es aquel que resulta totalmente distinto de todos los valores posibles de la lista que se está le-

[2] Estos métodos son descritos en Savitch, Walter, *Problem Solving with C++, The Object of Programming*, 2a. edición, Reading, Massachussetts, Addison-Wesley, 1999.

yendo, y de este modo sirve para indicar el final. Un ejemplo típico se presenta cuando al leer una lista de números positivos; un número negativo puede utilizarse como un valor centinela para indicar el final de la lista.

```
// ejemplo de valor centinela (número negativo)
...
cout << "Introduzca una lista de enteros positivos" << endl;
     << "Termine la lista con un número negativo" << endl;
suma = 0;
cin >> número;
while (número >= 0)
{
   suma = suma + número;
   cin >> número;
}
cout << "La suma es: " << suma;
```

Si al ejecutar el segmento de programa anterior se introduce la lista

```
4       8       15       -99
```

el valor de la suma será 27. Es decir, -99, último número de la entrada de datos no se añade a suma. -99 es el último dato de la lista que actúa como centinela y no forma parte de la entrada de números.

Agotamiento de la entrada

Cuando se leen entradas de un archivo, se puede utilizar un valor centinela. Aunque el método más frecuente es comprobar simplemente si todas las entradas del archivo se han leído, y se alcanza el final del bucle cuando no hay más entradas a leer. Éste es el método usual en la lectura de archivos, que suele utilizar una marca al final de archivo, `eof`.

Cristian Palapanta 1095211-2202110031

5.11. Estructuras repetitivas anidadas

De igual forma que se pueden anidar o encajar estructuras de selección, es posible insertar un bucle dentro de otro. Las reglas para construir estructuras repetitivas anidadas son iguales en ambos casos: la estructura interna debe estar incluida totalmente dentro de la externa y no puede existir solapamiento. La representación gráfica se indica en la figura 5.6.

Figura 5.6. Bucles anidados: *a*) y *b*) correctos, *c*) y *d*) incorrectos.

Las variables índices o de control de los bucles toman valores de modo tal que por cada valor de la variable índice del ciclo externo se debe ejecutar totalmente el bucle interno. Es posible anidar cualquier tipo de estructura repetitiva con tal que cumpla las condiciones de la figura 5.5.

EJEMPLO 5.16

Se conoce la población de cada una de las veinticinco ciudades más grandes de las ocho provincias de Andalucía, y se desea identificar y visualizar la población de la ciudad más grande de cada provincia.

El problema consistirá, en primer lugar, en la obtención de la población mayor de cada provincia y realizar esta operación ocho veces, una para cada provincia.

1. Encontrar y visualizar la ciudad mayor de una provincia.
2. Repetir el paso 1 para cada una de las ocho provincias andaluzas.

El procedimiento para deducir la ciudad más grande de entre las veinticinco de una provincia se consigue creando una variable auxiliar MAYOR (inicialmente de valor 0) que se va comparando sucesivamente con los veinticinco valores de cada ciudad, de modo tal que, según el resultado de comparación, se intercambian valores de la ciudad por el de la variable MAYOR. El algoritmo correspondiente sería:

```
algoritmo CIUDADMAYOR
var
   entero : i      //contador de provincias
   entero : j      //contador de ciudades
   entero : MAYOR  //ciudad de mayor población
   entero : CIUDAD //población de la ciudad
inicio
   i ← 1
   mientras i <= 8 hacer
      MAYOR ← 0
      j ← 1
      mientras j <= 25 hacer
         leer(CIUDAD)
         si CIUDAD > MAYOR entonces
            MAYOR ← CIUDAD
         fin_si
         j ← j + 1
      fin_mientras
      escribir('La ciudad mayor es', MAYOR)
      i ← i + 1
   fin_mientras
fin
```

EJEMPLO 5.17

Calcular el factorial de n números leídos del terminal.

El problema consistirá en realizar una estructura repetitiva de *n* iteraciones del algoritmo del problema ya conocido del cálculo del factorial de un entero.

```
algoritmo factorial2
var
```

Cristian Pilapanta 1096211-2202110032

```
   entero : i, NÚMERO, n
   real : FACTORIAL
inicio
   {lectura de la cantidad de números}
   leer(n)
   desde i ← 1 hasta n hacer
      leer(NÚMERO)
      FACTORIAL ← 1
      desde j ← 1 hasta NÚMERO hacer
         FACTORIAL ← FACTORIAL * j
      fin_desde
      escribir('El factorial del número', NÚMERO, 'es', FACTORIAL)
   fin_desde
fin
```

EJEMPLO 5.18

Imprimir todos los números primos entre 2 y 100 inclusive.

```
algoritmo Primos
var entero: i, divisor
   logico: primo
inicio
   desde i ← hasta 100 hacer
      primo ← verdad
      divisor ← 2
      mientras (divisor <= raíz2(i)) y primo hacer
         si i mod divisor = 0 entonces
            primo ← falso
         si_no
            divisor ← divisor + 1
         fin_si
      fin_mientras
      si primo entonces
         escribir(i,' ')
      fin_si
   fin_desde
fin
```

Cristian Pilapanta 1006211-22021110033

1006211220211003

5.11.1. Bucles (lazos) anidados: una aplicación en C++

Es posible *anidar* bucles. Los bucles anidados constan de un bucle externo con uno o más bucles internos. Cada vez que se repite el bucle externo, los bucles se repiten, se reevalúan los componentes de control, y se ejecutan todas las iteraciones requeridas.

EJEMPLO 5.19

El segmento de programa siguiente visualiza una tabla de multiplicación por cálculo y visualización de productos de la forma `x * y` *para cada* `x` *en el rango de 1 a* `Xultimo` *y desde cada* `y` *en el rango 1 a* `Yultimo` *(donde* `Xultimo`, `Yultimo` *son enteros prefijados). La tabla que se desea obtener es:*

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
        ...
```

```
for (int x = 1; x <= Xultimo; x++)

    for (int y = 1; y <= Yultimo; y++)
    {
       producto = x * y;
       cout << setw(2) << x << " * "
            << setw(2) << y << " = "
            << setw(3) << producto << endl;
    }
```

bucle externo *bucle interno*

El bucle que tiene x como variable de control se denomina **bucle externo**, y el que tiene y como variable de control se denomina **bucle interno**.

EJEMPLO 5.20

```
// Aplicación de bucles anidados
#include <iostream>
#include <iomanip.h>      // necesario para cin y cout
using namespace std;      // necesario para setw

void main()
{
   // cabecera de impresión
   cout << setw(12) << "i" << setw(6) << "j" << endl;
   for (int i = 0; i < 4; i++)
   {
      cout << "Externo" << setw(7) << i << endl;
      for (int j = 0; j < i; j++)
         cout << "Interno" << setw(10) << j << endl;
   } // fin del bucle externo
}
```

La salida del programa es:

```
                i      j
Externo         0
Externo         1
```

Cristian P Espana 1006211-2202110534

```
   Interno          0
Externo       2
   Interno          0
   Interno          1
Externo       3
   Interno          0
   Interno          1
   Interno          2
```

EJERCICIO 5.1

Escribir un programa que visualice un triángulo isósceles.

El triángulo isósceles se realiza mediante un bucle externo y dos internos. Cada vez que se repite el bucle externo se ejecutan los dos internos. El externo se repite cinco veces (cinco filas); el número de repeticiones realizadas por los internos se basa en el valor de la variable `fila`. El primer bucle interno visualiza los espacios en blanco no significativos; el segundo bucle interno visualiza uno o más asteriscos.

```
// archivo triángulo.cpp
#include <iostream>
using namespace std;
void main()
{
   // datos locales...
   const int num_lineas = 5;
   const char blanco = ' ';
   const char asterisco = '*';

   // comienzo de una nueva línea
   cout << endl;

   // dibujar cada línea: bucle externo
   for (int fila = 1; fila <= num_lineas; fila++)
   {
      // imprimir espacios en blanco: primer bucle interno
      for (int blancos = num_lineas -fila; blancos > 0;
         blancos--)
         cout << blanco;

      for (int cuenta_as = 1; cuenta_as < 2 * fila;
           cuenta_as ++)
           cout << asterisco;

      // terminar línea
      cout << endl;
   } // fin del bucle externo
}
```

Cristian Pilapanta 1086211-2202110235

El bucle externo se repite cinco veces, una por línea o fila; el número de repeticiones ejecutadas por los bucles internos se basa en el valor de `fila`. La primera fila consta de un asterisco y cuatro blancos, la fila 2 consta de tres blancos y tres asteriscos, y así sucesivamente; la fila 5 tendrá 9 asteriscos ($2 \times 5 - 1$).

EJERCICIO 5.2

Ejecutar y visualizar el programa siguiente que imprime una tabla de m filas por n columnas y un carácter prefijado.

```
1: //Listado
2: //ilustra bucles for anidados
3:
4: int main()
5: {
6:    int filas, columnas;
7:    char elCar;
8:    cout << "¿Cuántas filas?";
9:    cin >> filas;
10:   cout << "¿Cuántas columnas?";
11:   cin >> columnas;
12:   cout << "¿Qué carácter?";
13:   cin >> elCar;
14:   for (int i = 0; i < filas; i++)
15:   {
16:      for (int j = 0; j < columnas; j++)
17:         cout << elCar;
18:   cout << "\n";
19:   }
20:   return 0;
21: }
```

Cristian Pilapanta 1095211-2202110036

1095211220221100-36

5.12. Sentencias de salto en C++, Java y Python: `break`, `continue`, `return`, `goto`

Como ya se ha comentado, las sentencias de salto o bifurcación incondicional **`break`**, **`continue`** y **`return`** están incorporadas en la mayoría de los lenguajes de programación. En esta sección hacemos una breve revisión de las mismas. Demasiadas sentencias **`break`** y **`continue`** utilizadas en varios bucles pueden producir muchos puntos de salida y, en consecuencia, el programa puede ser difícil de leer.

Java

`break`

`Cuando` se ejecuta **`break`** en un bucle **`while`**, **`for`**, **`do-while`** o en la sentencia condicional **`switch`** produce la salida inmediata de la sentencia. La ejecución continúa en la primera sentencia después de la sentencia de control. Los usos comunes de **`break`** son "escapar" anticipadamente de un bucle o saltar el resto de una sentencia switch.

Formato: `break` *`etiqueta`*;

`Etiqueta` es el nombre de una etiqueta (cualquier identificador valido en Java) que identifica un bloque de código. Cuando se ejecuta la sentencia `break` el control se transfiere al bloque de código de ese nombre.

continue

La sentencia `continue` dentro de una sentencia de bucle, `while`, `for` o `do-while`, cuando se ejecuta salta las sentencias restantes del cuerpo del bucle y sigue con la próxima iteración del bucle. En las sentencias `while` y `do-while` el programa evalúa la prueba de continuación del bucle inmediatamente después de que se ejecuta la sentencia `continue`. En un bucle `for`, se ejecuta el incremento y a continuación se evalúa la prueba de continuación del bucle.

Formato: `break etiqueta;`

goto

Es una palabra reservada en Java, pero no está soportada la sentencia `goto`, aunque si pueden utilizarse rupturas etiquetadas en su lugar, en casos excepcionales. La etiqueta se utiliza para identificar un bloque de código.

C++

Bjarne Stroustrup el creador del lenguaje C++ en su obra *The C++ Programming Language* (3a. edición, 1997) explica los usos de las sentencias de salto **`break`**, **`continue`** y **`goto`**. En el apéndice A del libro, dedicado en exclusiva a describir la gramática del lenguaje C++ (págs. 793 a 815) y a la sintaxis de todas las palabras reservadas, sentencias, reglas gramaticales, expresiones, operadores, etc., y en la sección de sentencias, presenta la sintaxis de todas las sentencias de salto de C++:

```
break ;
continue ;
return expresión_opc   (opc, opcional, para indicar que puede o no haber un valor a devolver)
goto identificador ;
     identificador: sentencias
```

Nota relativa al uso de la sentencia `goto` de Stroupstrup (pág. 137 del libro de referencia de C++)
Stroustrup como es lógico no recomienda el uso de la sentencia **`goto`**, sin embargo, menciona algunos casos excepcionales donde sí se podría utilizar. Así señala: "la sentencia `goto` tiene pocos usos en programación de alto nivel, pero puede ser muy útil cuando el código de C++ se genera en un programa en lugar de ser escrito directamente por una persona; por ejemplo, las sentencias goto pueden ser útiles en un analizador (*parser*), generado desde una gramática por un generador de analizadores. También puede ser importante en los raros casos en que la eficiencia óptima es esencial; en el bucle interno de alguna aplicación de tiempo real" (se está refiriendo implícitamente a los compiladores o intérpretes de lenguajes). También señala otro de los pocos usos posibles de goto, ya en código ordinario: "salir de un bucle anidado o una sentencia switch (break sólo interrumpe el bucle interior que lo contiene o una sentencia `switch`)".

Python

El lenguaje Python soporta las sentencias **`break`** y **`continue`**, que funcionan igual que en el lenguaje C. No contiene en su conjunto de sentencias, a `goto`. La sentencia `break` termina el bucle (o lazo) `for` o `while` *más anidado. Por el contrario,* `continue` al ejecutarse continúa en la siguiente iteración del bucle.

`break`
Formato: **`break`**

```
x = 0
while x<10
   if x == 6
      break
   print x
   x = x+1
continue
```

El bucle anterior se interrumpe cuando x vale 6 (valor que no llegará a imprimirse) aunque se siga cumpliendo la condición de `while`.

Cristian Palapanta 1095211-2202110036

Continue

Formato: `continue`

```
x = 0
while x<10
   x = x+1
   if x == 6
      continue
   print x
```

Funciona igual que el bucle anterior con `break`, pero con un resultado diferente, se imprimen los números del 1 al 10 pero se omite el 6 (que ha saltado por la sentencia `continue`).

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

Cristian Pilapanta 1095211-2202110037

1095211-2202110037

5.1. *Calcular el factorial de un número N utilizando la estructura* **desde**.

Solución

Recordemos que factorial de `N` responde a la fórmula

`N! = N · (N − 1) · (N − 2) · (N − 3) · ... · 3 · 2 · 1`

El algoritmo `desde` supone conocer el número de iteraciones:

```
var
   entero : I, N
   real : FACTORIAL
inicio
   leer(N)
   FACTORIAL ← 1
   desde I ← 1 hasta N hacer
      FACTORIAL ← FACTORIAL * I
   fin_desde
   escribir('El factorial de', N, 'es', FACTORIAL)
fin
```

5.2. *Imprimir las treinta primeras potencias de 4, es decir, 4 elevado a 1, 4 elevado a 2, etcétera.*

Solución

```
algoritmo potencias4
var
   entero : n
inicio
   desde n ← 1 hasta 30 hacer
   escribir(4 ^ n)
   fin_desde
fin
```

5.3. *Calcular la suma de los n primeros números enteros utilizando la estructura* `desde`.

Solución

$S = 1 + 2 + 3 + \ldots + n$

El pseudocódigo correspondiente es:

```
algoritmo sumaNenteros
var
   entero : i, n
   real : suma
inicio
   leer(n)
   suma ← 0
   desde i ← 1 hasta n hacer
      suma ← suma + 1
   fin_desde
   {escribir el resultado de suma}
  escribir(suma)
fin
```

5.4. *Diseñar el algoritmo para imprimir la suma de los números impares menores o iguales que n.*

Solución

Los números impares son 1, 3, 5, 7, ..., *n*. El pseudocódigo es:

```
algoritmo sumaimparesmenores
var
   entero : i, n
   real : S
inicio
   S ← 0
   leer(n)
   desde i ← 1 hasta n inc 2 hacer
      S ← S + i
   fin_desde
   escribir(S)
fin
```

5.5. *Dados dos números enteros, realizar el algoritmo que calcule su cociente y su resto.*

Solución

Sean los números `M` y `N`. El método para obtener el cociente y el resto es por restas sucesivas; el método sería restar sucesivamente el divisor del dividendo hasta obtener un resultado menor que el divisor, que será el resto de la división; el número de restas efectuadas será el cociente

```
50      |13      50 - 13 = 37      C = 1
11      3        37 - 13 = 24      C = 2
                 24 - 13 = 11      C = 3
```

Como 11 es menor que el divisor 13, se terminarán las restas sucesivas y entonces 11 será el resto y 3 (número de restas) el cociente. Por consiguiente, el algoritmo será el siguiente:

```
algoritmo cociente
var
   entero : M, N, Q, R
inicio
   leer(M, N)    {M, dividendo / N, divisor}
   R ← M
```

Cristian Palpanta 1095211-2202110039

```
   Q ← 0
   repetir
      R ← R - N
      Q ← Q + 1
   hasta_que R < N
   escribir('dividendo',M, 'divisor',N, 'cociente',Q, 'resto',R)
fin
```

5.6. *Realizar el algoritmo para obtener la suma de los números pares hasta 1000.*

Solución

Método 1

S = 2 + 4 + 6 + 8 + ... 1000

```
algoritmo sumapares
var
   real : NÚMERO, SUMA
inicio
   SUMA ← 2
   NÚMERO ← 4
   mientras NÚMERO <= 1000 hacer
      SUMA ← SUMA + NÚMERO
      NÚMERO ← NÚMERO + 2
   fin_mientras
fin
```

Método 2

```
{idéntica cabecera y declaraciones}
inicio
   SUMA ← 2
   NÚMERO ← 4
   repetir
      SUMA ← SUMA + NÚMERO
      NÚMERO ← NÚMERO + 2
   hasta_que NÚMERO > 1000
fin
```

5.7. *Buscar y escribir la primera vocal leída del teclado. (Se supone que se leen, uno a uno, caracteres desde el teclado.)*

Solución

```
algoritmo buscar_vocal
var
   carácter: p
inicio
   repetir
      leer(p)
   hasta_que p = 'a' o p = 'e' o p = 'i' o p = 'o' o p = 'u'
   escribir('Primero', p)
fin
```

5.8. *Se desea leer de una consola a una serie de números hasta obtener un número inferior a 100.*

Solución

```
algoritmo menor_100
var
```

Cristian Pasquini 10652112202110048

```
   real : número
inicio
   repetir
      escribir('Teclear un número')
      leer(número)
   hasta_que número < 100
   escribir('El número es', número)
fin
```

5.9. *Escribir un algoritmo que permita escribir en una pantalla la frase '¿Desea continuar? S/N' hasta que la respuesta sea 'S' o 'N'.*

Solución

```
algoritmo SN
var
   carácter : respuesta
inicio
   repetir
      escribir('Desea continuar S/N')
      leer(respuesta)
   hasta_que(respuesta = 'S') o (respuesta = 'N')
fin
```

5.10. *Leer sucesivamente números del teclado hasta que aparezca un número comprendido entre 1 y 5.*

Solución

```
algoritmo número1_5
var
   entero : número
inicio
   repetir
      escribir('Número comprendido entre 1 y 5')
      leer(número)
   hasta_que(número >= 1) y (número <= 5)
   escribir('Número encontrado', número)
fin
```

5.11. *Calcular el factorial de un número n con métodos diferentes al ejercicio 5.1.*

Solución

$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$

es decir,

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
$4! = 4 \times 3 \times 2 \times 1 = 24$
$3! = 3 \times 2 \times 1 = 6$
$2! = 2 \times 1 = 2$
$1! = 1 = 1$

Para codificar estas operaciones basta pensar que

$$(n+1)! = (n+1) \times n \times \underbrace{(n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1}_{n!}$$

$(n+1)! = (n+1) \times n!$

Cristian Pliapanta 1095211-2202110040

Por consiguiente, para calcular el FACTORIAL de un número necesitaremos un contador i que cuente de uno en uno y aplicar la fórmula:

```
FACTORIAL = FACTORIAL * i
```

Inicializando los valores de FACTORIAL e i a 1 y realizando un bucle en el que i se incremente en 1 a cada iteración, es decir,

Algoritmo 1 de Factorial de n

```
FACTORIAL ← 1
i ← 1
repetir
   FACTORIAL ← FACTORIAL * i
   i ← i + 1
hasta_que i = n + 1
```

Algoritmo 2 de Factorial de n

```
FACTORIAL ← 1
i ← 1
repetir
   FACTORIAL ← FACTORIAL * (i + 1)
   i ← i + 1
hasta_que i = n
```

Algoritmo 3 de Factorial de n

```
FACTORIAL ← 1
i ← 1
repetir
   FACTORIAL ← FACTORIAL * (i + 1)
   i ← i + 1
hasta_que i > n - 1
```

Algoritmo 4 de factorial de n

```
FACTORIAL ← 1
i ← 1
desde i ← 1 hasta n - 1 hacer
FACTORIAL ← FACTORIAL * (i + 1)
fin_desde
```

Un algoritmo completo con lectura del número n por teclado podría ser el siguiente:

```
algoritmo factorial
var
   entero : i, n
   real : f
inicio
   f ← 1
   i ← 1
   leer(n)
   repetir
     f ← f * i
     i ← i + 1
   hasta_que i = n + 1
   escribir('Factorial de', n, 'es', f)
fin
```

Cristian Pilapanta 1006211-2202110041
1006211220211004

5.12. *Calcular el valor máximo de una serie de 100 números.*

Solución

Para resolver este problema necesitaremos un contador que cuente del 1 a 100 para contabilizar los sucesivos números. El algoritmo que calcula el valor máximo será repetido, y partiremos considerando que el primer número leído es el valor máximo, por lo cual se realizará una primera asignación del número 1 a la variable *máximo*.

La siguiente acción del algoritmo será realizar comparaciones sucesivas:

- Leer un nuevo número;
- Compararlo con el valor máximo;
- Si es *inferior*, implica que el valor máximo es el antiguo;
- Si es *superior*, implica que el valor máximo es el recientemente leído, por lo que éste se convertirá en *máximo* mediante una asignación;
- Repetir las acciones anteriores hasta que $n = 100$.

```
algoritmo máximo
var
   entero : n, número, máximo
inicio
   leer(número)
   n ← 1
   máximo ← número
   repetir
      n ← n+1
      leer(número)
      si número > máximo entonces
         máximo ← número
      fin_si
   hasta_que n = 100
   escribir('Número mayor o máximo', máximo)
fin
```

Otras soluciones

```
1. algoritmo otromaximo
     var
        entero : n, número, máximo
     inicio
        leer(número)
        máximo ← número
        n ← 2
        repetir
           n ← n + 1
           leer(número)
           si número > máximo entonces
              máximo ← número
           fin_si
        hasta_que n > 100
        escribir('Número mayor o máximo', máximo)
     fin
2. algoritmo otromaximo
     var
        entero : n, número, máximo
```

Cristian Pliapanta 1095211-2202110048

```
   inicio
      leer(número)
      máximo ← número
      para n = 2 hasta 100 hacer //pseudocódigo sustituto de desde
         leer(número)
         si número > máximo entonces
           máximo ← número
         fin_si
   fin_para
   escribir('Máximo,', máximo)
fin
```

Nota: Los programas anteriores suponen que los números pueden ser positivos o negativos; si se desea comparar sólo números positivos, los programas correspondientes serían:

```
1. algoritmo otromaximo
   var
      entero : n, número, máximo
   inicio
      n ← 0
      maximo ← 0
   repetir
      leer(número)
      n = n + 1
      si número > máximo entonces
         máximo ← número
      fin_si
   hasta_que n = 100
   escribir('Máximo número', máximo)
fin
2. algoritmo otromaximo
   var
      entero : n, número, máximo
   inicio
      n ←0
      máximo ← 0
      para N ← 1 hasta 100 hacer
         leer(número)
         si número > máximo entonces
            máximo ← número
      fin_si
   fin_para
   escribir('Máximo número =', máximo)
fin
```

5.13. *Bucles anidados. Las estructuras de control tipo bucles pueden anidarse internamente, es decir, se puede situar un bucle en el interior de otro bucle.*

Solución

La anidación puede ser:

- bucles **repetir** dentro de bucles **repetir**,

Cristian Pilapanta 1095211-2202110107
1095211220211010 7

- bucles **para (desde)** dentro de bucles **repetir**,
- etcétera.

Ejemplo 1. Bucle **para** en el interior de un bucle **repetir-hasta_que**

```
repetir
  leer(n)
  para i ← 1 hasta hacer 5
    escribir(n * n)
  fin_para
hasta_que n = 0
escribir('Fin')
```

Si ejecutamos estas instrucciones, se obtendrá para:

```
 n = 5        resultados          25
                                  25
                                  25
                                  25
                                  25
 n = 2        resultados          4
                                  4
                                  4
                                  4
                                  4

 n = 0        resultados          0
                                  0
                                  0
                                  0
                                  0
              fin
```

Ejemplo 2. Bucles anidados **para (desde)**

```
para i ← 1 hasta 3 hacer
   para j ← 1 hasta 10 hacer
      escribir(i, 'por', j, '=', i*j)
   fin_para
fin_para
```

Los valores sucesivos de `i, j, i*j,` serán

```
i = 1  j = 1      i * j = 1 x 1 = 1
       j = 2      i * j = 1 x 2 = 2
       j = 3      i * j = 1 x 3 = 3
       j = 4      i * j = 1 x 4 = 4
.....................................
       j = 10     i * j = 1 x 10 = 10
i = 2  j = 1      i * j = 2 x 1 = 2
       j = 2      i * j = 2 x 2 = 4
       j = 3      i * j = 2 x 3 = 6
       j = 4      i * j = 2 x 4 = 8
.....................................
       j = 10     i * j = 2 x 10 = 20
```

Cristian Pilapanta 1006211-2202110108

```
i = 3  j = 1     i * j = 3 x 1 = 3
       j = 2     i * j = 3 x 2 = 6
       j = 3     i * j = 3 x 3 = 9
       j = 4     i * j = 3 x 4 = 12
.....................................
       j = 10    i * j = 3 x 10 = 30
```

Es decir, se obtendrá la tabla de multiplicar de 1, 2 y 3.

Ejemplo 3. La codificación completa para obtener la popular tabla de multiplicar de los escolares sería la siguiente:

```
algoritmo Tabla_de_multiplicar
var
   entero : i, j, producto
inicio
   para ← 1 hasta 9 hacer
      escribir('Tabla del', i)
      para j ← 1 hasta 10 hacer
         producto ← i * j
         escribir(i, 'por', j, '=', producto)
      fin_para
   fin_para
fin
```

5.14. *Se dispone de una lista de* N *números. Se desea calcular el valor del número mayor.*

Solución

```
algoritmo
var
   entero : I
   real : NÚM, MÁX
   entero : N
inicio
   leer(N)
   leer(NÚM)
   MÁX ← NÚM
   desde I ← 2 hasta 100 hacer
      leer(NÚM)
      si NÚM > MÁX entonces
         MÁX ← NÚM
      fin_si
   fin_desde
fin
```

5.15. *Determinar simultáneamente los valores máximo y mínimo de una lista de 100 números.*

Solución

```
algoritmo máx_mín
var
   entero : I
   real : MÁX, MÍN, NÚMERO
inicio
   leer(NÚMERO)
   MÁX ← NÚMERO
```

Cristian Palpanta 1065211-2202110108
1065211220211010B

```
   MÍN ← NÚMERO
   desde I ← 2 hasta 100 hacer
      leer(NÚMERO)
      si NÚMERO > MÁX entonces
         MÁX ← NÚMERO
      si_no
         si NÚMERO < MÍN entonces
            MÍN ← NÚMERO
         fin_si
      fin_si
   fin_desde
   escribir('Máximo', MÁX, 'Mínimo', MÍN)
fin
```

5.16. *Se dispone de un cierto número de valores de los cuales el último es 999, y se desea determinar el valor máximo de las medias correspondientes a parejas de valores sucesivos.*

Solución

```
algoritmo media_parejas
var
   entero : N1, N2
   real : M, MÁX
inicio
   leer(N1, N2)
   MÁX ← (N1 + N2)/ 2
   mientras (N2 <> 999) o (N1 <> 999) hacer
      leer(N1, N2)
      M ← (N1 + N2)/2
      si M > MÁX entonces
         MÁX ← M
      fin_si
   fin_mientras
   escribir('Media máxima =' MÁX)
fin
```

5.17. *Detección de entradas numéricas —enteros— erróneas.*

Solución

Análisis

Este algoritmo es una aplicación sencilla de interruptor. Se sitúa el valor inicial del interruptor (`SW = 0`) antes de recibir la entrada de datos.

La detección de números no enteros se realizará con una estructura repetitiva **`mientras`** que se realizará si `SW = 0`.

La instrucción que detecta si un número leído desde el dispositivo de entradas es entero:

```
leer(N)
```

Realizará la comparación de `N` y parte entera de `N`:

- Si son iguales, `N` es entero.
- Si son diferentes, `N` no es entero.

Un método para calcular la parte entera es utilizar la función estándar **`ent(int)`** existente en muchos lenguajes de programación.

Cristian Palpanta 1065211-2202110109

Pseudocódigo

```
algoritmo error
var
   entero: SW
   real: N
inicio
   SW ← 0
   mientras SW = 0 hacer
      leer(N)
      si N <> ent(N) entonces
         escribir('Dato no válido')
         escribir('Ejecute nuevamente')
         SW ← 1
      si_no
         escribir('Correcto', N, 'es entero')
      fin_si
   fin_mientras
fin
```

5.18. *Calcular el factorial de un número dado (otro nuevo método).*

Solución

Análisis

El factorial de un número N (N!) es el conjunto de productos sucesivos siguientes:

$N! = N * (N - 1) * (N - 2) * (N - 3) * \ldots * 3 * 2 * 1$

Los factoriales de los primeros números son: $1! = 1$

$2! = 2 * 1 = 2 * 1!$
$3! = 3 * 2 * 1 = 3 * 2!$
$4! = 4 * 3 * 2 * 1 = 4 * 3!$
.
.
.
$N! = N * (N - 1) * (N - 2) * \ldots * 2 * 1 = N * (N - 1)!$

Los cálculos anteriores significan que el factorial de un número se obtiene con el producto del número N por el factorial de $(N - 1)!$

Como comienzan los productos en 1, un sistema de cálculo puede ser asignar a la variable *factorial* el valor 1. Se necesita otra variable I que tome los valores sucesivos de 1 a N para poder ir efectuando los productos sucesivos. Dado que en los números negativos no se puede definir el factorial, se deberá incluir en el algoritmo una condición para verificación de error, en caso de que se introduzcan números negativos desde el terminal de entrada ($N < 0$).

La solución del problema se realiza por dos métodos:

1. Con la estructura `repetir (repeat)`.
2. Con la estructura `desde (for)`.

Pseudocódigo

Método 1 (estructura `repetir`)

```
algoritmo FACTORIAL
var
   entero: I, N
   real: factorial
```

Cristian Pilapanta 1095211-2202110110

```
inicio
   repetir
      leer(N)
   hasta_que N > 0
   factorial ← 1
   I ← 1
   repetir
      factorial ← factorial * I
      I ← I + 1
   hasta_que I = N + 1
   escribir(factorial)
fin
```

Método 2 (estructura **desde**)

```
algoritmo FACTORIAL
var
   entero : K, N
   real : factorial
inicio
   leer(N)
   si n < 0 entonces
      escribir('El número será positivo')
   si_no
      factorial ← 1
      si N > 1 entonces
         desde K ← 2 hasta N hacer
            factorial ← factorial * K
         fin_desde
      fin_si
      escribir('Factorial de', N, '=', factorial)
   fin_si
fin
```

5.19. *Se tienen las calificaciones de los alumnos de un curso de informática correspondiente a las asignaturas Basic, Pascal, Fortran. Diseñar un algoritmo que calcule la media de cada alumno.*

Solución

Análisis

Asignaturas: C
Pascal
Fortran

$$\text{Media:} \quad \frac{(C + Pascal + Fortran)}{3}$$

Se desconoce el número de alumnos N de la clase; por consiguiente, se utilizará una marca final del archivo ALUMNOS. La marca final es '***' y se asignará a la variable *nombre*.

Cristian Pilapanta 1085211-2202110111

Pseudocódigo

```
algoritmo media
var
   cadena : nombre
   real : media
   real : C, Pascal, Fortran
inicio
   {entrada datos de alumnos}
   leer(nombre)
   mientras nombre <> '***' hacer
      leer(C, Pascal, Fortran)
      media ← (C + Pascal + Fortran)/3
      escribir(nombre, media)
      leer(nombre)
   fin_mientras
fin
```

Cristian Palpanta 1095211-22\02110112

CONCEPTOS CLAVE

- bucle (lazo)
- bucle (lazo) anidado
- bucle (lazo) infinito
- bucle (lazo) sin fin
- centinela
- iteración
- pasada
- programación estructurada
- sentencia `continuar`
- sentencia `ir_a`
- sentencia `interrumpir`
- sentencia `de repetición`
- sentencia `desde`
- sentencia `hacer-mientras`
- sentencia `mientras`
- sentencia `nula`
- sentencia `repetir-hasta_que`

RESUMEN

Este capítulo examina los aspectos fundamentales de la iteración y el modo de implementar esta herramienta de programación esencial utilizando los cuatro tipos fundamentales de sentencias de iteración: `mientras`, `hacer-mientras`, `repetir-hasta_que` y `desde (para)`.

1. Una sección de código repetitivo se conoce como bucle. El bucle se controla por una sentencia de repetición que comprueba una condición para determinar si el código se ejecutará. Cada pasada a través del bucle se conoce como una iteración o repetición. Si la condición se evalúa como falsa en la primera iteración, el bucle se termina y se habrán ejecutado las sentencias del cuerpo del bucle una sola vez. Si la condición se evalúa como verdadera la primera vez que se ejecuta el bucle, será necesario que se modifiquen algunas sentencias del interior del bucle para que se altere la condición correspondiente.
2. Existen cuatro tipos básicos de bucles: `mientras`, `hacer-mientras`, `repetir-hasta_que` y `desde`. Los bucles `mientras` y `desde` son bucles controlados por la entrada o pretest. En este tipo de bucles, la condición comprobada se evalúa al principio del bucle, que requiere que la condición sea comprobada explícitamente antes de la entrada al bucle. Si la condición es verdadera, las repeticio-

nes del bucle comienzan; en caso contrario, no se introduce al bucle. Las iteraciones continúan mientras que la condición permanece verdadera. En la mayoría de los lenguajes, estas sentencias se construyen utilizando las sentencias **while** y **for**, respectivamente. Los bucles **hacer-mientras** y **repetir- hasta_que** son bucles controlados *por salida* o *posttest*, en los que la condición a evaluar se comprueba al final del bucle. El cuerpo del bucle se ejecuta siempre al menos una vez. El bucle **hacer-mientras** se ejecuta siempre que la condición sea verdadera y se termina cuando la condición se hace falsa; por el contrario, el bucle **repetir-hasta que** se realiza siempre que la condición es falsa y se termina cuando la condición se hace verdadera.

3. Un bucle **mientras** es un bucle con condición de entrada, de modo que puede darse el caso de que su cuerpo de sentencias no se ejecute nunca si la condición es falsa en el momento de entrar al bucle. Por el contrario, los bucles **hacer-mientras** y **repetir-hasta_que** son bucles de salida y, por consiguiente, las sentencias del cuerpo del bucle al menos se ejecutarán una vez.
4. Los bucles también se clasifican en función de la condición probada. En un bucle de conteo fijo, la condición sirve para fijar cuantas iteraciones se realizarán. En un bucle con condición variable (**mientras**, **hacer-mientras** y **repetir-hasta_ que**), la condición comprobada está basada en que una variable puede cambiar interactivamente con cada iteración a través del bucle.
5. La sintaxis de la sentencia mientras es:

```
mientras   cuenta = 1
   <sentencias>   mientras (cuenta <= 10)
                  hacer
   cuenta = cuenta + 1
fin_mientras      fin_mientras
```

6. La sentencia desde (**for**) realiza las mismas funciones que la sentencia **mientras** pero utiliza un formato diferente. En muchas situaciones, especialmente aquellas que utilizan una condición de conteo fijo, la sentencia desde es más fácil de utilizar que la sentencia mientras **equivalente**.

```
desde v ← vi hasta of [inc/dec] hacer
   <sentencias>
fin_desde
```

7. La sentencia **hacer_mientras** se utiliza para crear bucles *posttest*, ya que comprueba su expresión al final del bucle. Esta característica asegura que el cuerpo de un bucle hacer se ejecute al menos una vez. Dentro de un bucle **hacer** debe haber al menos una sentencia que modifique el valor de la expresión comprobada.
8. La programación estructurada utiliza las sentencias explicadas en este capítulo. Esta programación se centra en el modo de escribir las partes detalladas de programas de una computadora como módulos independientes. Su filosofía básica es muy simple: "Utilice sólo construcciones que tengan un punto de entrada y un punto de salida". Esta regla básica se puede romper fácilmente si se utiliza la sentencia de salto **ir_a**, por lo que no es recomendable su uso, excepto en situaciones excepcionales.

EJERCICIOS

5.1. Determinar la media de una lista indefinida de números positivos, terminados con un número negativo.

5.2. Dado el nombre de un mes y si el año es o no bisiesto, deducir el número de días del mes.

5.3. Sumar los números enteros del 1 a 100 mediante: *a*) estructura **repetir**, *b*) estructura **mientras**; *c*) estructura **desde**.

5.4. Determinar la media de una lista de números positivos terminada con un número no positivo después del último número válido.

5.5. Imprimir todos los números primos entre 2 y 1000.

5.6. Se desea leer las calificaciones de una clase de informática y contar el número total de aprobados (5 o mayor que 5).

5.7. Leer las notas de una clase de informática y deducir todas aquellas que son NOTABLES (>= 7 y < 9).

5.8. Leer 100 números. Determinar la media de los números positivos y la media de los números negativos.

Cristian Pilapanta 1095211-2202110112

5.9. Un comercio dispone de dos tipos de artículos en fichas correspondientes a diversas sucursales con los siguientes campos:

- Código del artículo A o B.
- Precio unitario del artículo.
- Número de artículos.

La última ficha del archivo de artículos tiene un código de artículo, una letra X. Se pide:

- El número de artículos existentes de cada categoría,
- El importe total de los artículos de cada categoría.

5.10. Una estación climática proporciona un par de temperaturas diarias (máxima, mínima) (no es posible que alguna o ambas temperaturas sea 9 °C). La pareja fin de temperaturas es 0.0. Se pide determinar el número de días, cuyas temperaturas se han proporcionado, las medias máxima y mínima, el número de errores (temperaturas de 9 °C) y el porcentaje que representaban.

5.11. Calcular:

$$E(x) = 1 + x = \frac{x^2}{2!} + \ldots + \frac{n^n}{n!}$$

a) Para N que es un entero leído por teclado.
b) Hasta que N sea tal que $xn/n < E$ (por ejemplo, $E = 10^{-4}$).

5.12. Calcular el enésimo término de la serie de Fibonacci definida por:

$$A_1 = 1 \quad A_2 = 2 \quad A_3 = 1 + 2 = A_1 + A_2$$
$$A_n = A_{n-1} + A_{n-2} \ (n >= 3)$$

5.13. Se pretende leer todos los empleados de una empresa (situados en un archivo EMPRESA) y a la terminación de la lectura del archivo se debe visualizar un mensaje "existen trabajadores mayores de 65 años en un número de...", y el número de trabajadores mayores de 65 años.

5.14. Un capital C está situado a un tipo de interés R. ¿Al término de cuántos años se duplicará?

5.15. Se desea conocer una serie de datos de una empresa con 50 empleados: *a*) ¿Cuántos empleados ganan más de 300 000 pesetas al mes (salarios altos); *b*) entre 100 000 y 300 000 pesetas (salarios medios); y *c*) menos de 100 000 pesetas (salarios bajos y empleados a tiempo parcial)?

5.16. Imprimir una tabla de multiplicar como

	1	2	3	4	...	15
**	**	**	**	**	...	**
1*	1	2	3	4	...	15
2*	2	4	6	8	...	30
3*	3	6	9	12	...	45
4*	4	8	12	16	...	60
.						
.						
.						
15*	15	30	45	60	...	225

5.17. Dado un entero positivo n (> 1), comprobar si es primo o compuesto.

REFERENCIAS BIBLIOGRÁFICAS

DIJKSTRA, E. W.: "Goto Statement Considered Harmful", *Communications of the ACM*, vol. 11, núm. 3, marzo 1968, 147-148, 538, 541.

KNUTH, D. E.: "Structured Programming with goto Statements", *Computing Surveys*, vol. 6, núm. 4, diciembre 1974, 261.

Cristian Pulgarín 1095211-2202110114

CAPÍTULO 6

Subprogramas (subalgoritmos): funciones

6.1. Introducción a los subalgoritmos o subprogramas
6.2. Funciones
6.3. Procedimientos (subrutinas)
6.4. Ámbito: variables locales y globales
6.5. Comunicación con subprogramas: paso de parámetros
6.6. Funciones y procedimientos como parámetros
6.7. Los efectos laterales
6.8. Recursión (recursividad)
6.9. Funciones en C/C++, Java y C#
6.10. Ámbito (alcance) y almacenamiento en C/C++ y Java
6.11. Sobrecarga de funciones en C++ y Java
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Cristian Pilapanta 1095211-2202110115

INTRODUCCIÓN

La resolución de problemas complejos se facilita considerablemente si se dividen en problemas más pequeños (subproblemas). *La solución de estos subproblemas se realiza con subalgoritmos*, lo que le permite al programador desarrollar programas de problemas complejos utilizando el método descendente introducido en los capítulos anteriores. *Los subalgoritmos (subprogramas) pueden ser de dos tipos: funciones y procedimientos o subrutinas*. Los subalgoritmos son unidades de programa o módulos que están diseñados para ejecutar alguna tarea específica. Estas funciones y procedimientos se escriben solamente una vez, pero pueden ser referenciados en diferentes puntos de un programa, de modo que se puede evitar la duplicación innecesaria del código.

Las unidades de programas en el estilo de programación modular son independientes; el programador puede escribir cada módulo y verificarlo sin preocuparse de los detalles de otros módulos. Esto facilita considerablemente la localización de un error cuando se produce. Los programas desarrollados de este modo normalmente son más fáciles de comprender, ya que la estructura de cada unidad de programa se puede estudiar independientemente de las otras unidades de programa. En este capítulo se describen las *funciones y procedimientos*, junto con los conceptos de *variables locales y globales*, así como *parámetros*. Se introduce también el concepto de *recursividad* como una nueva herramienta de resolución de problemas.

6.1. Introducción a los subalgoritmos o subprogramas

Un método ya citado para solucionar un problema complejo es dividirlo en subproblemas —problemas más sencillos— y a continuación dividir estos subproblemas en otros más simples, hasta que los problemas más pequeños sean fáciles de resolver. Esta técnica de dividir el problema principal suele denominarse *divide y vencerás (divide and conquer)*. Este método de diseñar la solución de un problema principal obteniendo las soluciones de sus subproblemas se conoce como *diseño descendente (top-down design)*; se denomina así debido a que se inicia en la parte superior con un problema general y el diseño específico de las soluciones de los subproblemas. Normalmente las partes en que se divide un programa deben poder desarrollarse independientemente entre sí.

Las soluciones de un diseño descendente pueden implementarse fácilmente en lenguajes de programación de alto nivel, como C/C++, Python o Fortran. Estas partes independientes se denominan *subprogramas* o *subalgoritmos* si se emplean desde el concepto algorítmico.

La correspondencia entre el diseño descendente y la solución por computadora en términos de programa principal y sus subprogramas se analizará a lo largo de este capítulo.

Consideremos el problema del cálculo de la superficie (área) de un rectángulo. Este problema se puede dividir en tres subproblemas:

```
subproblema 1: entrada de datos de altura y base.
subproblema 2: cálculo de la superficie.
subproblema 3: salida de resultados.
```

El algoritmo correspondiente que resuelve los tres *subproblemas* es:

```
leer (altura, base)                  //entrada de datos
área ← base * altura                 //cálculo de la superficie
escribir(base, altura, área)         //salida de resultados
```

El método descendente se muestra en la figura 6.1.

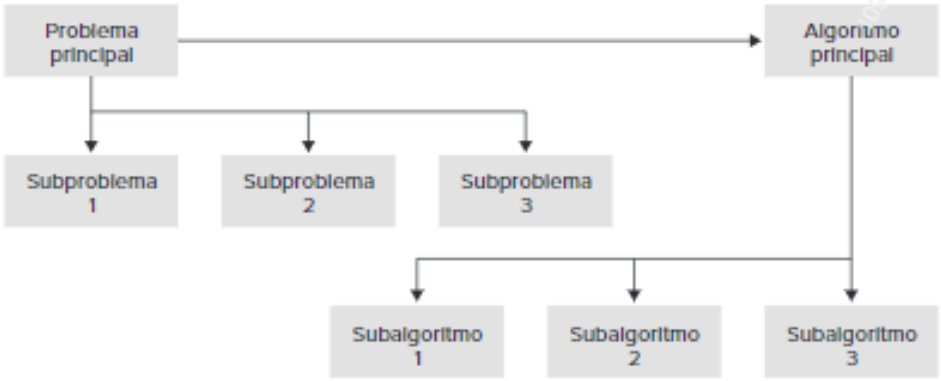


Figura 6.1. Diseño descendente.

El problema principal se soluciona por el correspondiente **programa** o **algoritmo principal** —también denominado *controlador o conductor* (*driver*)— y la solución de los subproblemas mediante **subprogramas**, conocidos como **procedimientos (subrutinas)** o **funciones**. Los subprogramas, cuando se tratan en lenguaje algorítmico, se denominan también *subalgoritmos*.

Un subprograma puede realizar las mismas acciones que un programa: 1) aceptar datos, 2) realizar algunos cálculos y 3) devolver resultados. Sin embargo, el programa utiliza un subprograma para un propósito específico; el subprograma recibe datos desde el programa y le devuelve resultados. Haciendo un símil con una oficina, el problema es como el jefe que da instrucciones a sus subordinados —subprogramas—; cuando la tarea se termina, el subordinado devuelve sus resultados al jefe. Se dice que el programa principal *llama* o *invoca* al subprograma y este ejecuta una tarea, a continuación *devuelve* el control al programa. Esto puede suceder en diferentes lugares del pro-

Cristian Pilapanta 1095211-2202110116

grama. Cada vez que se llama al subprograma, el control retorna al lugar desde donde fue hecha la llamada (figura 6.2); un subprograma puede llamar a su vez a sus propios subprogramas (figura 6.3). Existen —como ya se ha comentado— dos tipos importantes de subprogramas: *funciones* y *procedimientos o subrutinas*.

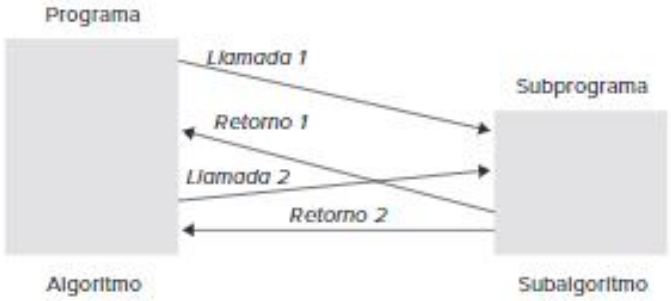


Figura 6.2. Un programa con un subprograma: función y procedimiento o subrutina, según la terminología específica del lenguaje: subrutina en Fortran, *función* en C, C++, Python método en Java o C#, *procedimiento* o *función* en Pascal.

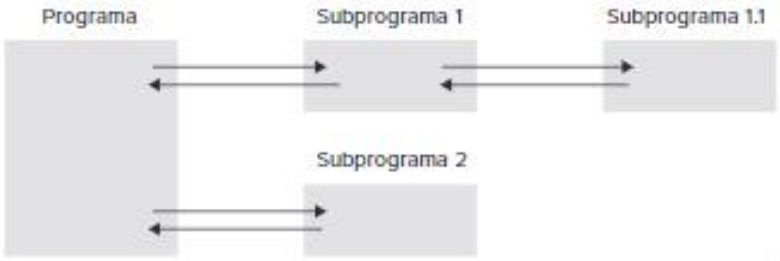


Figura 6.3. Un programa con diferentes niveles de subprogramas.

Cristian Pilapanta 1095211-2202110116

6.2. Funciones

Matemáticamente una función es una operación que toma uno o más valores llamados *argumentos* y produce un valor denominado *resultado* —valor de la función para los argumentos dados—. Todos los lenguajes de programación tienen *funciones incorporadas, intrínsecas* o *internas*, y *funciones definidas por el usuario*. Así, por ejemplo:

$$f(x) = \frac{x}{1 + x^2}$$

donde f es el nombre de la función y x es el argumento. Obsérvese que ningún valor específico se asocia con x, es un *parámetro formal* utilizado en la definición de la función. Para evaluar f debemos darle un *valor real o actual* a x, con este valor se puede calcular el resultado. Con $x = 3$ se obtiene el valor 0.3, que se expresa escribiendo:

$$f(3) = 0.3$$

$$f(3) = \frac{3}{1 + 9} = \frac{3}{10} = 0.3$$

Una función puede tener varios argumentos. Por consiguiente:

$$f(x, y) = \frac{x - y}{\sqrt{x} + \sqrt{y}}$$

es una función con dos argumentos. Sin embargo, solamente un único valor se asocia con la función para cualquier par de valores dados a los argumentos.

Cada lenguaje de programación tiene sus propias funciones incorporadas, que se utilizan escribiendo sus nombres con los argumentos adecuados en expresiones tales como:

```
raiz2(A+cos(x))
```

Cuando la expresión se evalúa, el valor de *x* se da primero al subprograma (función) coseno y se calcula *cos(x)*. El valor de *A+cos(x)* se utiliza entonces como argumento de la función *raiz2* (raíz cuadrada), que evalúa el resultado final.

Cada función se evoca utilizando su nombre en una expresión con los argumentos actuales o reales encerrados entre paréntesis.

Las funciones incorporadas al sistema se denominan *funciones internas o intrínsecas* y definidas por el usuario, *funciones externas*. Cuando las funciones estándares o internas no permiten realizar el tipo de cálculo deseado es necesario recurrir a las funciones externas que pueden ser definidas por el usuario mediante una *declaración de función*.

A una función no se le llama explícitamente, sino que se le invoca o referencia mediante un nombre y una lista de parámetros actuales. El algoritmo o programa llama o invoca a la función con el nombre de esta última en una expresión seguida de una lista de argumentos que deben coincidir en cantidad, tipo y orden con los de la función que fue definida. La función devuelve un único valor.

Las funciones son diseñadas para realizar tareas específicas: toman una lista de valores —llamados *argumentos*— y devuelven un único valor.

Cristian Pachano 1005211-2202110118

6.2.1. Declaración de funciones

La declaración de una función requiere una serie de pasos que la definen. Una función como tal subalgoritmo o subprograma tiene una constitución similar a los algoritmos, por consiguiente, constará de una cabecera que comenzará con el tipo del valor devuelto, seguido de la palabra **función** y del nombre y argumentos de dicha función. A continuación irá el *cuerpo*, que será una serie de acciones o instrucciones cuya ejecución hará que se asigne un valor al nombre de la función. Esto determina el valor particular del resultado que ha de devolverse al programa llamador.

La declaración de la función será:

```
<tipo_de_resultado> función <nombre_fun> (lista de parámetros)
[declaraciones locales]
inicio
   <acciones>          //cuerpo de la función
   devolver (<expresión>)
fin_función
```

`lista de parámetros`	lista de *parámetros formales* o *argumentos*, con uno o más argumentos de la siguiente forma:
	`([E|S|E/S] tipo_de_datoA: parámetro 1[, parámetro 2]...;`
	`[E|S|E/S] tipo_de_datoB: parámetro x[, parámetro y]...)`

nombre_func	*nombre asociado con la función, que será un nombre de identificador válido*
<acciones>	instrucciones que constituyen la definición de la función y que debe contener una única instrucción: **devolver** (<expresión>); *expresión* sólo existe si la función se ha declarado con valor de retorno y *expresión* es el valor devuelto por la función
tipo_de_resultado	tipo del resultado que devuelve la función

Sentencia `devolver (return)`

La sentencia **`devolver`** (**`return`**, volver) se utiliza para regresar de una función (un *método* en programación orientada a objetos); **`devolver`** hace que el control del programa se transfiera al llamador de la función (método). Esta sentencia se puede utilizar para hacer que la ejecución regrese de nuevo al llamador de la función.

Regla

La sentencia `devolver` termina inmediatamente la función en la cual se ejecuta.

Por ejemplo, la función:

$$f(x) = \frac{x}{1 + x^2}$$

se definirá como:

```
real función F(E real:x)
inicio
   devolver (x/(1+x*x))
fin_función
```

Otro ejemplo puede ser la definición de la función trigonométrica, cuyo valor es:

$$tan(x) = \frac{sen(x)}{cos(x)}$$

donde *sen*(*x*) y *cos*(*x*) son las funciones seno y coseno —normalmente funciones internas—. La declaración de la función es:

```
real función tan (E real:x)
//función tan igual a sen(x)/cos(x), ángulo x en radianes
inicio
   devolver (sen(x)/cos(x))
fin_función
```

Observe que se incluye un comentario para describir la función. Es buena práctica incluir documentación que describa brevemente lo que hace la función, lo que representan sus parámetros o cualquier otra información que la explique o la defina. *En aquellos lenguajes de programación —como Pascal— que exigen sección de declaraciones,* éstas *se situarán al principio de la función.*

Para que las acciones descritas en un subprograma función sean ejecutadas, se necesita que éste sea invocado desde un programa principal o desde otros subprogramas, a fin de proporcionarle los argumentos de entrada necesarios para realizar esas acciones.

Los argumentos de la declaración de la función se denominan *parámetros formales, ficticios o mudos* (*dummy*); son nombres de variables, de otras funciones o procedimientos y que sólo se utilizan dentro del cuerpo de la función. Los argumentos utilizados en llamada a la función se denominan *parámetros actuales*, que a su vez pueden ser constantes, variables, expresiones, valores de funciones o nombres de funciones o procedimientos.

6.2.2. Invocación a las funciones

Una función puede ser llamada de la forma siguiente:

```
nombre_función              (lista de parámetros actuales)
```

`nombre_función`	función que llama
`lista de parámetros actuales`	constantes, variables, expresiones, valores de funciones, nombres de funciones o procedimientos

Cristian Pilapanta 1095211-2202110118

Cada vez que se llama a una función desde el algoritmo principal se establece automáticamente una correspondencia entre los parámetros formales y los parámetros actuales. Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales en la declaración de la función y se presupone una correspondencia uno a uno de izquierda a derecha entre los parámetros formales y los actuales.

Una llamada a la función implica los siguientes pasos:

1. A cada parámetro formal se le asigna el valor real de su correspondiente parámetro actual.
2. Se ejecuta el cuerpo de acciones de la función.
3. Se devuelve el valor de la función y se retorna al punto de llamada.

EJEMPLO 6.1

Definir la función: $y = x^n$ (potencia n de x).

```
real : función potencia(E real:x;E entero:n)
var
   entero: i, y
inicio
   y ← 1
   desde i ← 1 hasta abs(n) hacer
      y ← y*x
   fin_desde
   si n < 0 entonces
      y ← 1/y
   fin_si
   devolver (y)
fin_función
```

abs(n) es la función valor absoluto de *n* a fin de considerar exponentes positivos o negativos.

Invocación de la función

```
z ← potencia (2.5, -3)
```

parámetros actuales

Transferencia de información

```
x = 2.5    n = -3
z = 0.064
```

EJEMPLO 6.2

Función potencia para el cálculo de N elevado a. El número N deberá ser positivo, aunque podrá tener parte fraccionaria, A es un real.

```
algoritmo Elevar_a_potencia
var
   real : a, n
inicio
   escribir('Deme número positivo')
   leer(n)
   escribir('Deme exponente')
   leer(a)
   escribir('N elevado a =', potencia(n, a))
fin
real función potencia (E real: n, a)
```

Cristian Pilapanta 1006211-2202110120

```
inicio
   devolver(EXP(a * LN(n)))
fin_función
```

EJEMPLO 6.3

Diseñar un algoritmo que contenga un subprograma de cálculo del factorial de un número y una llamada al mismo.

Como ya es conocido por el lector el algoritmo factorial, lo indicaremos expresamente:

```
entero función factorial(E entero:n)
var
   entero: i,f
   //advertencia, según el resultado, f puede ser real
inicio
   f ← 1
   desde i ← 1 hasta n hacer
      f ← f * i
   fin_desde
   devolver (f)
fin_función
```

y el algoritmo que contiene un subprograma de cálculo del factorial de un número y una llamada al mismo:

```
algoritmo función_factorial
var entero: x, y, número

inicio
   escribir ('Deme un número entero y positivo')
   leer(número)
   x ← factorial(número)
   y ← factorial(5)
   escribir(x, y)
fin
```

En este caso los parámetros actuales son: una variable (*número*) y una constante (*5*).

EJEMPLO 6.4

Realizar el diseño de la función $y = x^3$ *(cálculo del cubo de un número).*

```
algoritmo prueba
var
   entero: N
inicio     //Programa principal
   N ← cubo(2)
   escribir ('2 al cubo es', N)
   escribir ('3 al cubo es', cubo(3))
fin

entero función cubo(E entero: x)
inicio
   devolver(x*x*x)
fin_función
```

La salida del algoritmo sería:

```
2 al cubo es 8
3 al cubo es 27
```

Krasher Padres 1095564-2202171623

Las funciones pueden tener muchos argumentos, pero solamente un resultado: *el valor de la función*. Esto limita su uso, aunque se encuentran con frecuencia en cálculos científicos. Un concepto más potente es el proporcionado por el subprograma procedimiento que se examina en el siguiente apartado.

EJEMPLO 6.5

Algoritmo que contiene y utiliza unas funciones (seno y coseno) a las que les podemos pasar el ángulo en grados.

```
algoritmo Sen_cos_en_grados
var real : g

inicio
   escribir('Deme ángulo en grados')
   leer(g)
   escribir(seno(g))
   escribir(coseno(g))
fin

real función coseno (E real : g)
inicio
   devolver(COS(g*2*3.141592/360))
fin_función

real: función seno (E real g)
inicio
   devolver(SEN(g*2*3.141592/360))
fin_función
```

EJEMPLO 6.6

Realizar un algoritmo que simplifique un quebrado, dividiendo numerador y denominador por su máximo común divisor.

```
algoritmo Simplificar_quebrado
var
entero : n, d
inicio
   escribir('Deme numerador')
   leer(n)
   escribir('Deme denominador')
   leer(d)
   escribir(n, '/', d, '=', n div mcd(n, d),'/', d div mcd(n, d))
fin
   entero función mcd (E entero: n, d)
   var
      entero : r
   inicio
      r ← n MOD d
      mientras r <> 0 hacer
         n ← d
         d ← r
         r ← n MOD d
      fin_mientras
      devolver(d)
   fin_función
```

Kristher Padres 1095584-2202171623

EJEMPLO 6.7

Supuesto que nuestro compilador no tiene la función seno. Podríamos calcular el seno de x mediante la siguiente serie:

$$\text{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \text{ (hasta 17 términos)}$$

x (ángulo en radianes).

El programa nos tiene que permitir el cálculo del seno de ángulos en grados mediante el diseño de una función seno(x), que utilizará, a su vez, las funciones `potencia(x,n)` y `factorial(n)`, que también deberán ser implementadas en el algoritmo.

Se terminará cuando respondamos `N (no)` a la petición de otro ángulo.

```
algoritmo Calcular_seno
var real: gr
      carácter: resp
inicio
   repetir
      escribir('Deme ángulo en grados')
      leer(gr)
      escribir('Seno(', gr,')=', seno(gr))
      escribir('¿Otro ángulo?')
      leer(resp)
   hasta_que resp = 'N'
fin

real función factorial (E entero:n)
var
   real: f
   entero: i
inicio
   f ← 1
   desde i ← 1 hasta n hacer
   f ← f * i
   fin_desde
   devolver(f)
fin_función
real función potencia (E real:x; E entero:n)
var real : pot
   entero : i
inicio
   pot ← 1
   desde i ← 1 hasta n hacer
   pot ← pot * x
   fin_desde
   devolver(pot)
fin_función

real función seno (E real:gr)
var real : x, s
   entero : i, n
inicio
   x ← gr * 3.141592 / 180
   s ← x
   desde i ← 2 hasta 17 hacer
```

Krasher Padres 1095584-2202171623

```
        n ← 2 * i - 1
        si i MOD 2 <> 0 entonces
           s ← s - potencia(x, n) / factorial(n)
        si_no
           s ← s + potencia(x, n) / factorial(n)
        fin_si
     fin_desde
     devolver(s)
  fin_función
```

6.3. Procedimientos (subrutinas)

Aunque las funciones son herramientas de programación muy útiles para la resolución de problemas, su alcance está muy limitado. Con frecuencia se requieren subprogramas que calculen varios resultados en vez de uno solo, o que realicen la ordenación de una serie de números, etcétera. En estas situaciones la *función* no es apropiada y se necesita disponer del otro tipo de subprograma: el *procedimiento* o *subrutina*.

Un *procedimiento o subrutina*[1] es un subprograma que ejecuta un proceso específico. Ningún valor está asociado con el nombre del procedimiento; por consiguiente, no puede ocurrir en una expresión. Un procedimiento se llama escribiendo su nombre, por ejemplo, `SORT`, para indicar que un procedimiento denominado `SORT (ordenar)` se va a usar. Cuando se invoca el procedimiento, los pasos que lo definen se ejecutan y a continuación se devuelve el control al programa que le llamó.

Kristine Podnes 1095584-2202171624

Procedimiento *versus* función

Los procedimientos y funciones son subprogramas cuyo diseño y misión son similares; sin embargo, existen unas diferencias esenciales entre ellos.

1. Un procedimiento es llamado desde el algoritmo o programa principal mediante su nombre y una lista de parámetros actuales, o bien con la instrucción **`llamar_a (call)`**. Al llamar al procedimiento se detiene momentáneamente el programa que se estuviera realizando y el control pasa al procedimiento llamado. Después que las acciones del procedimiento se ejecutan, se regresa a la acción inmediatamente siguiente a la que se llamó.
2. Las funciones devuelven un valor, los procedimientos pueden devolver 0,1 o n valores y en forma de lista de parámetros.
3. El procedimiento se declara igual que la función, pero su nombre no está asociado con ninguno de los resultados que obtiene.

La *declaración de un procedimiento* es similar a la de funciones.

```
procedimiento nombre [(lista de parámetros formales)]
   <acciones>
fin_procedimiento
```

Los parámetros formales tienen el mismo significado que en las funciones; los parámetros variables –en aquellos lenguajes que los soportan, por ejemplo, Pascal– están precedidos cada uno de ellos por la palabra **`var`** para designar que obtendrán resultados del procedimiento en lugar de los valores actuales asociados con ellos.

[1] En Fortran, la subrutina representa el mismo concepto que procedimiento. No obstante, en la mayor parte de los lenguajes el término general para definir un subprograma es procedimiento o simplemente subprograma.

El procedimiento se llama mediante la instrucción siguiente:

```
[llamar_a] nombre [(lista de parámetros actuales)]
```

La palabra **llamar_a (call)** es opcional y su existencia depende del lenguaje de programación.

El siguiente ejemplo ilustra la definición y uso de un procedimiento para realizar la división de dos números y obtener el cociente y el resto.

Variables enteras:

```
Dividendo
Divisor
Cociente
Resto
```

Procedimiento

```
procedimiento división (E entero: Dividendo, Divisor; S entero: Cociente, Resto)
inicio
   Cociente ← Dividendo DIV Divisor
   Resto ← Dividendo - Cociente * Divisor
fin_procedimiento
```

Algoritmo principal

```
algoritmo aritmética
var
   entero: M, N, P, Q, S, T
inicio
   leer(M, N)
   llamar_a división (M, N, P, Q)
   escribir(P, Q)
   llamar_a división (M * N - 4, N + 1, S, T)
   escribir(S, T)
fin
```

6.3.1. Sustitución de argumentos/parámetros

Los parámetros, bien *formales* en el procedimiento o *actuales* (reales) en la llamada se conoce como *lista de parámetros*.

```
procedimiento demo
.
.
.
fin_procedimiento
```

o bien:

```
procedimiento demo (lista de parámetros formales)
```

y la instrucción que llama:

```
llamar_a demo (lista de parámetros actuales)
```

Krasher Padres 1095564-2202171625

Cuando se llama al procedimiento, cada parámetro formal toma como valor inicial el valor del correspondiente parámetro actual. En el ejemplo siguiente se indican la sustitución de parámetros y el orden correcto.

```
algoritmo demo
   //definición del procedimiento
      entero: años
      real: números, tasa
inicio
   ...
   llamar_a cálculo(número, años, tasa)
   ...
fin

procedimiento cálculo(S real: p1; E entero: p2; E real: p3)
inicio
   p3 ... p1 ... p2
fin_procedimiento
```

Las acciones sucesivas a realizar son:

1. Los parámetros reales sustituyen a los formales.
2. El cuerpo de la declaración del procedimiento se sustituye por la llamada del procedimiento.
3. Por último, se ejecutan las acciones escritas por el código resultante.

Kraiher Podres 1095584-2202171626

EJEMPLO 6.8

(DE PROCEDIMIENTO)

Diseñar un algoritmo que transforme un número introducido por teclado en notación decimal a romana. El número será entero y positivo y no excederá de 3.000.

1095584-2202171626

Sin utilizar programación modular

```
algoritmo romanos
var entero : n,dígito,r,j

inicio
   repetir
      escribir('Deme número')
      leer(n)
   hasta_que (n >= 0) Y (n <= 3000)
   r ← n
   dígito ← r DIV 1000
   r ← r MOD 1000
   desde j ← 1 hasta dígito hacer
      escribir('M')
   fin_desde
   dígito ← r DIV 100
   r ← r MOD 100
   si dígito = 9 entonces
      escribir('C', 'M')
   si_no
      si dígito > 4 entonces
         escribir('D')
         desde j ← 1 hasta dígito - 5 hacer
```

```
            escribir('C')
         fin_desde
      si_no
         si dígito = 4 entonces
            escribir('C','D')
         si_no
            desde j ← 1 hasta dígito hacer
               escribir('C')
            fin_desde
         fin_si
      fin_si
   fin_si
   dígito ← r DIV 10
   r ← r MOD 10
   si dígito = 9 entonces
      escribir('X', 'C')
         si_no
      si dígito > 4 entonces
         escribir('L')
         desde j ← 1 hasta dígito - 5 hacer
         escribir('X')
         fin_desde
      si_no
         si dígito = 4 entonces
            escribir('X','L')
         si_no
            desde j ← 1 hasta dígito hacer
               escribir('X')
            fin_desde
         fin_si
      fin_si
   fin_si
   dígito ← r
   si dígito = 9 entonces
      escribir('I', 'X')
   si_no
      si dígito > 4 entonces
         escribir('V')
         desde j ← 1 hasta dígito - 5 hacer
            escribir('I')
         fin_desde
      si_no
         si dígito = 4 entonces
            escribir('I','V')
         si_no
            desde j ← 1 hasta dígito hacer
               escribir('I')
            fin_desde
         fin_si
      fin_si
   fin_si
fin
```

Krasher Padres 1095564-2202171722

1095564220217172

Mediante programación modular

```
algoritmo Romanos
var entero : n, r, dígito
inicio
   repetir
      escribir('Deme número')
      leer(n)
   hasta_que (n >= 0) Y (n <= 3000)
   r ← n
   dígito ← r Div 1000
   r ← r MOD 1000
   calccifrarom(dígito, 'M', ' ',' ')
   dígito ← r Div 100
   r ← r MOD 100
   calccifrarom(dígito, 'C', 'D', 'M')
   dígito ← r Div 10
   r ← r MOD 10
   calccifrarom(dígito, 'X', 'L', 'C')
   dígito ← r
   calccifrarom(dígito, 'I', 'V', 'X')
fin
procedimiento calccifrarom(E entero: dígito; E carácter: v1, v2, v3)
var entero: j
inicio
   si dígito = 9 entonces
      escribir(v1, v3)
   si_no
      si dígito > 4 entonces
         escribir(v2)
         desde j ← 1 hasta dígito - 5 hacer
            escribir(v1)
         fin_desde
      si_no
         si dígito = 4 entonces
            escribir(v1, v2)
         si_no
            desde j ← 1 hasta dígito hacer
               escribir(v1)
            fin_desde
         fin_si
      fin_si
   fin_si
fin_procedimiento
```

6.4. Ámbito: variables locales y globales

Las variables utilizadas en los programas principales y subprogramas se clasifican en dos tipos:

- Variables locales.
- Variables globales.

Krasher Padres 1095584-2202171926

Una *variable local* es aquella que está declarada y definida dentro de un subprograma, en el sentido de que está dentro de ese subprograma y es distinta de las variables con el mismo nombre declaradas en cualquier parte del programa principal. *El significado de una variable se confina al procedimiento en el que está declarada.* Cuando otro subprograma utiliza el mismo nombre se refiere a una posición diferente en memoria. Se dice que tales variables son *locales* al subprograma en el que están declaradas.

Una *variable global* es aquella que está declarada para el programa o algoritmo principal, del que dependen todos los subprogramas.

La parte del programa/algoritmo en que una variable se define se conoce como ámbito o *alcance* (*scope*). El uso de variables locales tiene muchas ventajas. En particular, hace a los subprogramas independientes, con la comunicación entre el programa principal y los subprogramas manipulados estructuralmente a través de la lista de parámetros. Para utilizar un procedimiento sólo necesitamos conocer lo que hace y no tenemos que estar preocupados por su diseño, es decir, cómo están programados.

Esta característica hace posible dividir grandes proyectos en piezas más pequeñas independientes. Cuando diferentes programadores están implicados, ellos pueden trabajar independientemente.

A pesar del hecho importante de los subprogramas independientes y las variables locales, la mayoría de los lenguajes proporcionan algún método para tratar ambos tipos de variables (figura 6.4).

Una variable local a un subprograma no tiene ningún significado en otros subprogramas. Si un subprograma asigna un valor a una de sus variables locales, este valor no es accesible a otros programas, es decir, no pueden utilizar este valor. A veces, también es necesario que una variable tenga el mismo nombre en diferentes subprogramas.

Por el contrario, las variables globales tienen la ventaja de compartir información de diferentes subprogramas sin una correspondiente entrada en la lista de parámetros.

En un programa sencillo con un subprograma, cada variable u otro identificador es o bien local al procedimiento o global al programa completo. Sin embargo, si el programa incluye procedimientos que engloban a otros procedimientos —*procedimientos anidados*—, entonces la noción de global/local es algo más complicado de entender.

Krasher Padres 1095564-2202171926

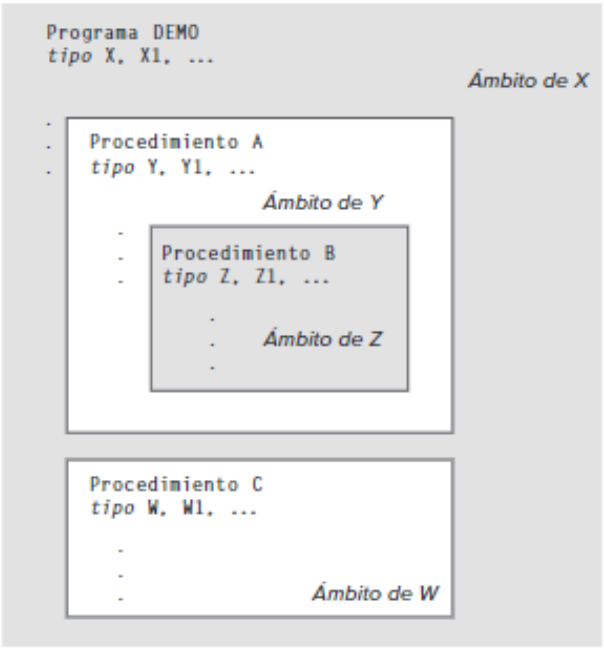
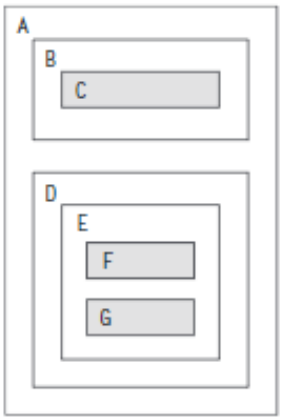


Figura 6.4. Ámbito de identificadores.

El *ámbito* de un identificador (variables, constantes, procedimientos) es la parte del programa donde se conoce el identificador. Si un procedimiento está definido localmente a otro procedimiento, tendrá significado sólo dentro

del ámbito de ese procedimiento. A las variables les sucede lo mismo; si están definidas localmente dentro de un procedimiento, su significado o uso se confina a cualquier función o procedimiento que pertenezca a esa definición.

La figura 6.5 muestra un esquema de un programa con diferentes procedimientos, algunas variables son locales y otras globales. En la citada figura se muestra el ámbito de cada definición.



Variables definidas en	Accesibles desde
A	A, B, C, D, E, F, G
B	B, C
C	C
D	D, E, F, G
E	E, F, G
F	F
G	G

Figura 6.5. Ámbito de definición de variables.

Krasner Podres 1095584-2202171927

Los lenguajes que admiten variables locales y globales suelen tener la posibilidad explícita de definir dichas variables como tales en el cuerpo del programa, o, lo que es lo mismo, definir su ámbito de actuación, para ello se utilizan las cabeceras de programas y subprogramas, con lo que se definen los ámbitos.

Las variables definidas en un ámbito son accesibles en el mismo, es decir, en todos los procedimientos interiores.

EJEMPLO 6.9

La función (signo) realiza la siguiente tarea: dado un número real x, si x es 0, entonces se devuelve un 0; si x es positivo, se devuelve 1, y si x es negativo, se devuelve un valor –1.

La declaración de la función es:

```
entero función signo(E real: x)
var entero:s
inicio
   //valores de signo: +1,0,-1
   si x = 0 entonces s ← 0
   si x > 0 entonces s ← 1
   si x < 0 entonces s ← -1
   devolver (s)
fin_función
```

Antes de llamar a la función, la variable (S), como se declara dentro del subprograma, es local y sólo se conoce dentro del mismo. Veamos ahora un pequeño algoritmo donde se invoque la función.

```
algoritmo SIGNOS
var
   entero: a, b, c
   real: x, y, z
```

```
inicio
   x ← 5.4
   a ← signo(x)
   y ← 0
   b ← signo(y)
   z ← 7.8975
   c ← signo(z - 9)
   escribir('Las respuestas son', a, ' ', b, ' ', c)
fin
```

Si se ejecuta este algoritmo, se obtienen los siguientes valores:

```
x = 5.4                 x es el parámetro actual de la primera llamada a signo(x)
a = signo(5.4)          a toma el valor 1
y = 0
b = signo(0)            b toma el valor de 0
z = 7.8975
c = signo(7.8975-9)     c toma el valor –1
```

La línea escrita al final será:

```
Las respuestas son 1 0 -1
```

Krasner Podres 1095584-2202171927

EJEMPLO 6.10

```
algoritmo DEMOX
var entero: A, X, Y
inicio
   x ← 5
   A ← 10
   y ← F(x)
   escribir (x, A, y)
fin
entero función F(E entero: N)
var
   entero: X
inicio
   A ← 5
   X ← 12
   devolver(N + A)
fin_función
```

A la variable global `A` se puede acceder desde el algoritmo y desde la función. Sin embargo, `X` identifica a dos variables distintas: una local al algoritmo y sólo se puede acceder desde él y otra local a la función.

Al ejecutar el algoritmo se obtendrían los siguientes resultados:

```
X = 5
A = 10
Y = F(5)                invocación a la función F(N) se realiza un paso del parámetro actual X al
                        parámetro formal N
A = 5                   se modifica el valor de A en el algoritmo principal por ser A global
X = 12                  no se modifica el valor de X en el algoritmo principal porque X es local
F = 5+5 = 10            se pasa el valor del argumento X(5) a través del parámetro N
Y = 10
```

se escribirá la línea

```
5 5 10
```

ya que X es el valor de la variable local X en el algoritmo; A, el valor de A en la función, ya que se pasa este valor al algoritmo; Y es el valor de la función F(X).

6.5. Comunicación con subprogramas: paso de parámetros

Cuando un programa llama a un subprograma, la información se comunica a través de la lista de parámetros y se establece una correspondencia automática entre los parámetros formales y actuales. *Los parámetros actuales son "sustituidos" o "utilizados" en lugar de los parámetros formales.*

La declaración del subprograma se hace con:

```
procedimiento nombre (clase tipo_de_dato:  F1;
                      clase tipo_de_dato:  F2;
                      ......................
                      clase tipo_de_dato : Fn)
.
.
.
fin_procedimiento
```

y la llamada al subprograma con:

```
    llamar_a nombre (A1, A2, ..., An)
```

donde F1, F2, ..., Fn son los parámetros formales y A1, A2, ..., An los parámetros actuales o reales.
Las clases de parámetros podrían ser:

```
(E) Entrada
(S) Salida
(E/S) Entrada/Salida
```

Existen dos métodos para establecer la correspondencia de parámetros:

1. *Correspondencia posicional,* que se establece aparejando los parámetros reales y formales según su posición en las listas: así, Fi se corresponde con A_i, donde i = 1, 2, ..., n. Este método tiene algunas desventajas de legibilidad cuando el número de parámetros es grande.
2. *Correspondencia por el nombre explícito,* también llamado *método de paso de parámetros por nombre*. En este método, en las llamadas se indica explícitamente la correspondencia entre los parámetros reales y formales. Este método se utiliza en **Ada**. Un ejemplo sería:

   ```
   SUB(Y => B, X => 30);
   ```

   que hace corresponder el parámetro actual B con el formal Y, y el parámetro actual 30 con el formal X durante la llamada de SUB.

Por lo general, la mayoría de los lenguajes usan exclusivamente la correspondencia posicional y ese será el método empleado en este libro.

Las cantidades de información que pueden pasarse como parámetros son *datos de tipos simples, estructurados,* —en los lenguajes que admiten su declaración— y *subprogramas*.

Krasher Podres 1095584-22021171928

6.5.1. Paso de parámetros

Existen diferentes métodos para la *transmisión o el paso de parámetros* a subprogramas. Es preciso conocer el método adoptado por cada lenguaje, ya que la elección puede afectar a la semántica del lenguaje. Dicho de otro modo, un mismo programa puede producir diferentes resultados bajo diferentes sistemas de paso de parámetros.

Los parámetros pueden ser clasificados como:

Entradas: las entradas proporcionan valores desde el programa que llama y que se utilizan dentro de un procedimiento. En los subprogramas función, las entradas son los argumentos en el sentido tradicional;

Salidas: las salidas producen los resultados del subprograma; de nuevo si se utiliza el caso de una función, éste devuelve un valor calculado por dicha función, mientras que con procedimientos pueden calcularse cero, una o varias salidas;

Entradas/salidas: un solo parámetro se utiliza para mandar argumentos a un programa y para devolver resultados.

Desgraciadamente, el conocimiento del tipo de parámetros no es suficiente para caracterizar su funcionamiento; por ello, examinaremos los diferentes métodos que se utilizan para pasar o transmitir parámetros.

Los métodos más empleados para realizar el paso de parámetros son:

- *Paso por valor* (también conocido por *parámetro valor*).
- *Paso por referencia o dirección* (también conocido por *parámetro variable*).
- *Paso por nombre.*
- *Paso por resultado.*

Krasher Padres 1095564-2202171929

6.5.2. Paso por valor

El paso por valor se utiliza en muchos lenguajes de programación; por ejemplo, C, Modula-2, Pascal, Algol y Snobol. La razón de su popularidad es la analogía con los argumentos de una función, donde los valores se proporcionan en el orden de cálculo de resultados. Los parámetros se tratan como variables locales y los valores iniciales se proporcionan copiando los valores de los correspondientes argumentos.

Los parámetros formales –locales a la función– reciben como valores iniciales los valores de los parámetros actuales y con ello se ejecutan las acciones descritas en el subprograma.

No se hace diferencia entre un argumento que es variable, constante o expresión, ya que sólo importa el valor del argumento. La figura 6.6 muestra el mecanismo de paso por valor de un procedimiento con tres parámetros.

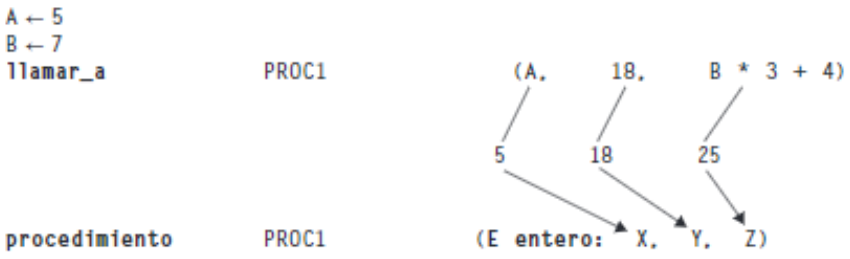


Figura 6.6. Paso por valor.

El mecanismo de paso se resume así:

Valor primer parámetro: `A = 5`.
Valor segundo parámetro: `constante = 18`.
Valor tercer parámetro: expresión `B * 3 + 4 = 25`.

Los valores `5`, `18` y `25` se transforman en los parámetros `X`, `Y`, `Z`, respectivamente cuando se ejecuta el procedimiento.

Aunque el *paso por valor* es sencillo, tiene una limitación acusada: *no existe ninguna otra conexión con los parámetros actuales*, y entonces los cambios que se produzcan por efecto del subprograma no producen cambios en los argumentos originales y, por consiguiente, no se pueden pasar valores de retorno al punto de llamada: es decir, todos los *parámetros* son sólo de *entrada*. El parámetro actual no puede modificarse por el subprograma. Cualquier cambio realizado en los valores de los parámetros formales durante la ejecución del subprograma se destruye cuando se termina el subprograma.

La llamada por valor no devuelve información al programa que llama.

Existe una variante de la llamada por valor y es la llamada por *valor resultado*. Las variables indicadas por los parámetros formales se inicializan en la llamada al subprograma por valor tras la ejecución del subprograma; los resultados (valores de los parámetros formales) se transfieren a los actuales. Este método se utiliza en algunas versiones de Fortran.

6.5.3. Paso por referencia

En numerosas ocasiones se requiere que ciertos parámetros sirvan como parámetros de salida, es decir, se devuelvan los resultados a la unidad o programas que llaman. Este método se denomina *paso por referencia* o también de *llamada por dirección o variable*. La unidad que llama pasa a la unidad llamada la dirección del parámetro actual (que está en el ámbito de la unidad llamante). Una referencia al correspondiente parámetro formal se trata como una referencia a la posición de memoria, cuya dirección se ha pasado. Entonces una variable pasada como parámetro real es compartida, es decir, se puede modificar directamente por el subprograma.

Este método existe en Fortran, Cobol, Modula-2, Pascal, PL/1 y Algol 68. Las características principales de este método son su simplicidad y su analogía directa con la idea de que las variables tienen una posición de memoria asignada desde la cual se pueden obtener o actualizar sus valores.

El área de almacenamiento (direcciones de memoria) se utiliza para pasar información de entrada y/o salida, en ambas direcciones.

En este método los parámetros son de entrada/salida y se denominan *parámetros variables*.

Los parámetros valor y variables suelen definirse en la cabecera del subprograma. En el caso de lenguajes como Pascal, los parámetros variables deben ir precedidos por la palabra clave **var**:

```
program muestra;
//parámetros actuales a, c, b y d paso por referencia
   procedure prueba(var x,y:integer);
   begin //procedimiento
      //proceso de los valores de x e y
   end;
begin
   .
   .
   .
1. prueba(a, c);
   .
   .
   .
2. prueba(b, d);
   .
   .
   .
end.
```

La primera llamada en (1) produce que los parámetros *a* y *c* sean sustituidos por *x* y *y* si estos valores se modifican dentro de *a* o *c* en el algoritmo principal. De igual modo, *x* y *y* sustituyen a *b* y *d*, y cualquier modificación de *x* o *y* en el procedimiento afectará también al programa principal.

La llamada por referencia es muy útil para programas donde se necesita la comunicación del valor en ambas direcciones.

Krasner Podres 1095564-2202171930

Notas

Ambos métodos de paso de parámetros se aplican tanto a la llamada de funciones como a las de procedimientos:
- Una función tiene la posibilidad de devolver los valores al programa principal de dos formas: *a*) como valor de la función, o *b*) por medio de argumentos gobernados por la llamada de referencia en la correspondencia parámetro actual-parámetro formal.
- Un procedimiento sólo puede devolver valores por el método de devolución de resultados.

El lenguaje Pascal permite que el programador especifique el tipo de paso de parámetros y, en un mismo subprograma, unos parámetros se pueden especificar por valor y otros por referencia.

```
procedure Q(i:integer; var j:integer);
begin
   i := i+10;
   j := j+10;
   write(i, j)
end;
```

Los parámetros formales son `i`, `j`, donde `i` se pasa por valor y `j` por referencia.

6.5.4. Comparaciones de los métodos de paso de parámetros

Para examinar de modo práctico los diferentes métodos, consideremos un ejemplo único y veamos los diferentes valores que toman los parámetros. El algoritmo correspondiente con un procedimiento `SUBR`:

```
algoritmo DEMO
var
   entero: A,B,C
inicio   //DEMO
   A ← 3
   B ← 5
   C ← 17
   llamar_a SUBR(A, A, A + B, C)
   escribir(C)
fin      //DEMO
procedimiento SUBR (<Modo> entero: x, y;
                    E entero:z; <Modo> entero: v)
inicio
   x ← x+1
   v ← y+z
fin_procedimiento
```

Modo por valor

a) sólo por valor

no se transmite ningún resultado, por consiguiente

`C no varía C = 17`

b) valor_resultado

					x	= A	=	3
A	=	3			y	= A	=	3
B	=	5	*pasa al procedimiento*		z	= A	+	B = 8
C	=	17			v	= C	=	17

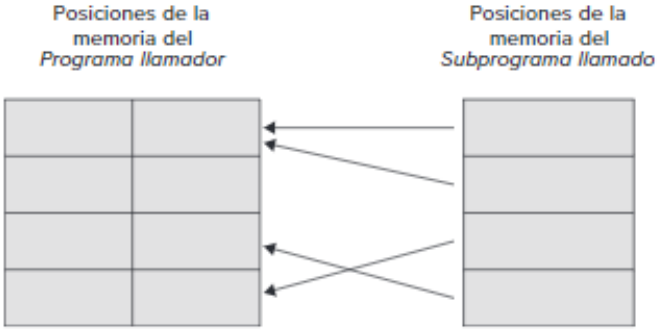
Krasher Padres 1095584-2202171931

Al ejecutar el procedimiento quedará:

```
x = x + 1 = 3 + 1 = 4
v = y + z = 3 + 8 = 11
```

El parámetro llamado *v* pasa el valor del resultado *v* a su parámetro actual correspondiente, `C`. Por tanto, `C` = 11.

Modo por referencia



`C` recibirá el valor 12.

Utilizando variables globales

```
algoritmo DEMO
var entero: A,B,C
inicio
   A ← 3
   B ← 5
   c ← 17
   llamar_a SUBR
   escribir (c)
fin

procedimiento SUBR
inicio
   a ← a + 1
   c ← a + a + b

fin_procedimiento
```

Es decir, el valor de `C` será 13.

La llamada por referencia es el sistema estándar utilizado por Fortran para pasar parámetros. La llamada por `nombre` es estándar en Algol 60. Simula 67 proporciona llamadas por valor, referencia y **`nombre`**.

Pascal permite pasar bien por valor bien por referencia:

```
procedure demo(y:integer; var z:real);
```

especifica que *y* se pasa por valor mientras que *z* se pasa por referencia –indicado por la palabra reservada **`var`**–. La elección entre un sistema u otro puede venir determinado por diversas consideraciones, como evitar efectos laterales no deseados provocados por modificaciones inadvertidas de parámetros formales (véase la sección 6.7).

Krasher Padres 1095564-2202171932

6.5.5. Síntesis de la transmisión de parámetros

Los métodos de transmisión de parámetros más utilizados son *por valor* y *por referencia.*

El paso de un parámetro por valor significa que el valor del argumento —*parámetro actual o real*— se asigna al parámetro formal. En otras palabras, antes de que el subprograma comience a ejecutarse, el argumento se evalúa a un valor específico (por ejemplo, 8 o 12), este valor se copia entonces en el correspondiente parámetro formal dentro del subprograma.

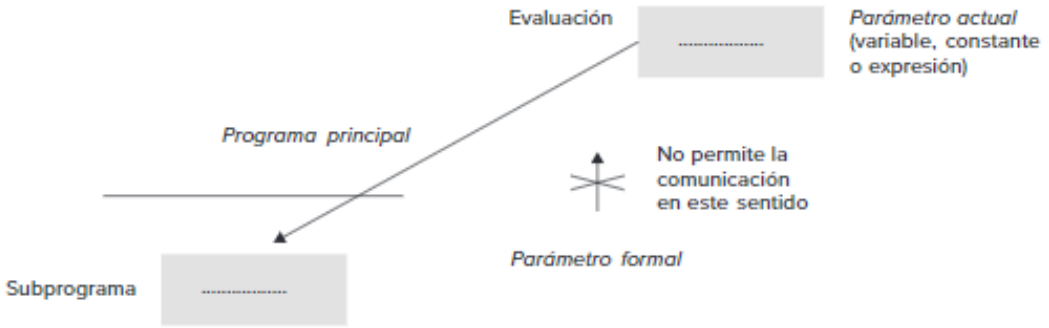


Figura 6.7. Paso de un parámetro por valor.

Krasher Padres 1095564-2202171933

Una vez que el procedimiento arranca, cualquier cambio del valor de tal parámetro formal no se refleja en un cambio en el correspondiente argumento. Esto es, cuando el subprograma termine, el argumento actual tendrá exactamente el mismo valor que cuando el subprograma comenzó, con independencia de lo que haya sucedido al parámetro formal. Éste es el método predeterminado en Pascal si no se indica explícitamente otro. Estos parámetros de entrada se denominan *parámetros valor* y en los algoritmos lo indicaremos como `<modo>` `E` (entrada). El paso de un *parámetro por referencia o dirección* se llama *parámetro variable*, en oposición al parámetro por valor. En este caso, la posición o dirección (no el valor) del argumento o parámetro actual se envía al subprograma. Si a un parámetro formal se le da el atributo de parámetro variable —en Pascal con la palabra reservada `var`— y si el parámetro actual es una variable, entonces un cambio en el parámetro formal se refleja en un cambio en el correspondiente parámetro actual, ya que ambos tienen la misma posición de memoria.

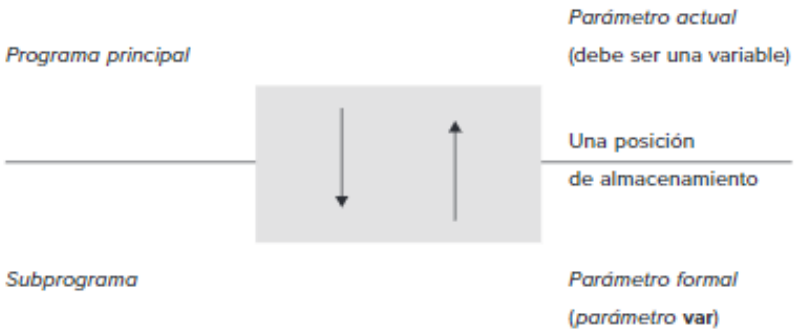


Figura 6.8. Paso de un parámetro por referencia.

Para indicar que deseamos transmitir un parámetro por dirección, utilizaremos la palabra *parámetro variable* —en Pascal se usa la palabra reservada `var`— y especificaremos como *<modo> E/S (entrada/salida) o S (salida).*

EJEMPLO 6.11

Realizar el cálculo del área de un círculo y la longitud de la circunferencia en función del valor del radio leído desde el teclado.

Recordemos las fórmulas del área del círculo y de la longitud de la circunferencia:

$A = pi.r2 = pi.r.r$

$C = 2.pi.r = 2.pi.r$ *donde* pi = 3.141592

Los parámetros de entrada: radio
Los parámetros de salida: área, longitud
El procedimiento círculo calcula los valores pedidos.

```
procedimiento círculo(E real: radio; S real: área, longitud)
   //parámetros valor: radio
   //parámetros variable: área, longitud
var
   real: pi
inicio
   pi ← 3.141592
   área ← pi * radio * radio
   longitud ← 2 * pi * radio
fin_procedimiento
```

Los parámetros formales son: radio, área, longitud, de los cuales son de tipo valor (radio) y de tipo variable (área, longitud).

Invoquemos el procedimiento círculo utilizando la instrucción

```
llamar_a círculo(6, A, C)
   //programa principal
inicio
   //llamada al procedimiento
   llamar_a círculo(6, A, C)
   .
   .
   .
fin

procedimiento círculo(E real: radio; S real: área, longitud)
//parámetros valor:radio
//parámetros variable: área, longitud

inicio
   pi ← 3.141592
   área ← pi * radio * radio
   longitud ← 2 * pi * radio
fin_procedimiento
```

EJEMPLO 6.12

Consideremos un subprograma M *con dos parámetros formales:* i, *transmitido por valor, y* j, *por variable.*

```
algoritmo M
//variables A, B enteras
var
   entero: A, B
inicio
   A ← 2
   B ← 3
   llamar_a N(A,B)
```

Krasher Padres 1095564-2202171934

```
   escribir(A, B)
fin    //algoritmo M
procedimiento N(E entero: i; E/S entero: j)
   //parámetros valor i
   //parámetros variable j
inicio
   i ← i + 10
   j ← j + 10
   escribir(i, j)
fin_procedimiento
```

Si se ejecuta el procedimiento `N`, veamos qué resultados se escribirán:

`A` y `B` son parámetros actuales.
`i` y `j` son parámetros formales.

Como i es por valor, se transmite el valor de `A` a i, es decir, $i = A = 2$. Cuando i se modifica por efecto de `i ← i+10` a `12`, `A` no cambia y, por consiguiente, a la terminación de `N`, `A` sigue valiendo `2`.

El parámetro `B` se transmite por referencia, es decir, j es un parámetro variable. Al comenzar la ejecución de `N`, `B` se almacena como el valor j y cuando se suma `10` al valor de j, i en sí mismo no cambia. El valor del parámetro `B` se cambia a `13`. Cuando los valores i, j se escriben en `N`, los resultados son:

```
12 y 13
```

Pero cuando retornan a `M` y al imprimir los valores de `A` y `B`, sólo ha cambiado el valor `B`. El valor de $i = 12$ se pierde en `N` cuando éste ya termina. El valor de j también se pierde, pero éste es la dirección, no el valor 13.
Se escribirá como resultado final de la instrucción ***escribir***`(A, B)`:

```
2  13
```

6.6. Funciones y procedimientos como parámetros

Hasta ahora los subprogramas que hemos considerado implicaban dos tipos de parámetros formales: *parámetros valor* y *parámetros variable*. Sin embargo, en ocasiones se requiere que un procedimiento o función dado invoque a otro procedimiento o función que ha sido definido fuera del ámbito de ese procedimiento o función. Por ejemplo, se puede necesitar que un procedimiento `P` invoque la función `F` que puede estar o no definida en el procedimiento `P`; esto puede conseguirse transfiriendo como parámetro el procedimiento o función externa `(F)` o procedimiento o función dado (por ejemplo, el `P`). En resumen, algunos lenguajes de programación –entre ellos Pascal– admiten *parámetros procedimiento* y *parámetros función*.

EJEMPLOS

```
procedimiento P(E func: F1; E real: x, y)
real función F(E func: F1, F2; E entero: x, y)
```

Los parámetros formales del procedimiento `P` son la función `F1` y las variables `x` y `y`, y los parámetros formales de la función `F` son las funciones `F1` y `F2`, y las variables `x` y `y`.

Procedimientos función

Para ilustrar el uso de los parámetros función, consideremos la función *integral* para calcular el área bajo una curva $f(x)$ para un intervalo $a <= x <= b$.

Krasner Padres 1095564-2202171935

La técnica conocida para el cálculo del área es subdividir la región en rectángulos, como se muestra en la figura 6.9, y sumar las áreas de los rectángulos. Estos rectángulos se construyen subdividiendo el intervalo [a, b] en *m* subintervalos iguales y formando rectángulos con estos subintervalos como bases y alturas dadas por los valores de *f* en los puntos medios de los subintervalos.

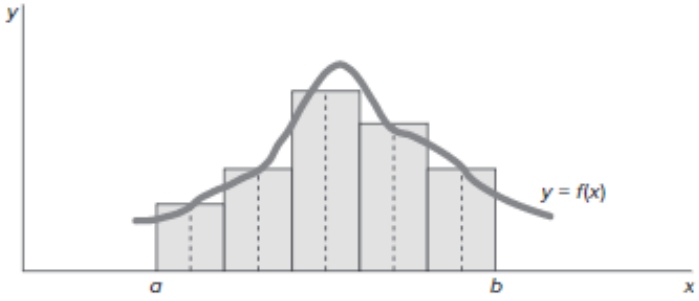


Figura 6.9. Cálculo del área bajo la curva *f*(*x*).

La función *integral* debe tener los parámetros formales *a*, *b* y *n*, que son valor ordinarios actuales de tipo real; éstos se asocian con los parámetros formales *a* y *b*; un parámetro actual de tipo entero —las subdivisiones— se asocia con el parámetro formal *n* y una función actual se asocia con el parámetro formal *f*.

Los parámetros función se designan como tales con una cabecera de función dentro de la lista de parámetros formales. La función integral podrá definirse por:

```
real función integral(E func: f; E real: a,b; E entero: n)
el tipo func-tipo real func función (E real: x)
```

aquí la función `f(x: real): real` especifica que *f* es una función parámetro que denota una función cuyo parámetro formal y valor son de tipo real. El correspondiente parámetro función actual debe ser una función que tiene un parámetro formal real y un valor real. Por ejemplo, si `Integrando` es una función de valor real con un parámetro y tipo real `función (E real:x):func`

```
Área ← Integral(Integrando, 0, 1.5, 20)
```

es una referencia válida a función.

Diseñar un algoritmo que utilice la función integral para calcular el área bajo el gráfico de las funciones $f1(x) = x^3 - 6x^3 + 10x$ y $f2(x) = x^2 + 3x + 2$ *para* $0 <= x <= 4$.

```
algoritmo Área_bajo_curvas

tipo
   real función(E real : x) : func
var
   real   : a,b
   entero : n
inicio
   escribir('¿Entre qué límites?')
   leer(a, b)
   escribir('¿Subintervalos?')
   leer(n)
   escribir(integral(f1, a, b, n))
   escribir(integral(f2, a, b, n))
fin
```

Krasner Podres 1095584-2202171936

```
real FUNCIÓN f1 (E real : x)
inicio
   devolver(x * x * x - 6 * x * x + 10 * x)
fin_función

real FUNCIÓN f2 (E real : x)
inicio
   devolver(x * x + 3 * x + 2)
fin_función

real FUNCIÓN integral (E func : f; E real : a, b; E entero : n)
var
   real : baserectángulo,altura,x,s
   entero : i
inicio
   baserectángulo ← (b - a) / n
   x ← a + baserectángulo/2
   s ← 0
   desde i ← 1 hasta n hacer
      altura ← f(x)
      s ← s + baserectángulo * altura
      x ← x + baserectángulo
   fin_desde
   devolver(s)
fin_función
```

Krasner Podres 1095564-2202171937

6.7. Los efectos laterales

Las modificaciones que se produzcan mediante una función o procedimiento en los elementos situados fuera del subprograma (función o procedimiento) se denominan *efectos laterales*. Aunque en algunos casos los efectos laterales pueden ser beneficiosos en la programación, es conveniente no recurrir a ellos de modo general. Consideramos a continuación los efectos laterales en funciones y en procedimientos.

6.7.1. En procedimientos

La *comunicación* del procedimiento con el resto del programa se debe realizar normalmente a través de parámetros. Cualquier otra *comunicación* entre el procedimiento y el resto del programa se conoce como *efectos laterales*. Como ya se ha comentado, los efectos laterales son perjudiciales en la mayoría de los casos, como se indica en la figura 6.10.

Si un procedimiento modifica una variable global (distinta de un parámetro actual), éste es un *efecto lateral*. Por ello, excepto en contadas ocasiones, no debe aparecer en la declaración del procedimiento. Si se necesita una variable temporal en un procedimiento, utilice una variable local, no una variable global. Si se desea que el programa modifique el valor de una variable global, utilice un parámetro formal variable en la declaración del procedimiento y a continuación utilice la variable global como el parámetro actual en una llamada al procedimiento.

En general, se debe seguir la regla de *ninguna variable global en procedimientos*, aunque esta prohibición no significa que los procedimientos no puedan manipular variables globales. De hecho, el cambio de variables globales se debe pasar al procedimiento como parámetros actuales. Las variables globales no se deben utilizar directamente en las instrucciones en el cuerpo de un procedimiento; en su lugar, utilice un parámetro formal o variable local.

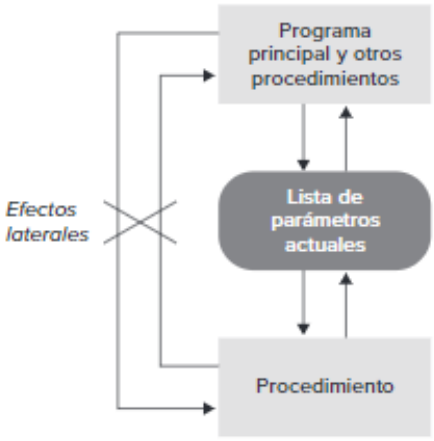


Figura 6.10. Efectos laterales en procedimientos.

En aquellos lenguajes en que es posible declarar constantes —como Pascal— se pueden utilizar constantes globales en una declaración de procedimiento; esto se debe al hecho de que las constantes no pueden ser modificadas por el procedimiento y, por consiguiente, no existe peligro de que se puedan modificar inadvertidamente.

Kraiher Pedres 1095584-2202171938

6.7.2. En funciones

Una función toma los valores de los argumentos y devuelve *un único valor*. Sin embargo, al igual que los procedimientos, una función —en algunos lenguajes de programación— puede hacer cosas similares a un procedimiento o subrutina. Una función puede tener parámetros variables además de parámetros valor en la lista de parámetros formales. Una función puede cambiar el contenido de una variable global y ejecutar instrucciones de entrada/salida (escribir un mensaje en la pantalla, leer un valor del teclado, etcétera). Estas operaciones se conocen como *parámetros laterales* y se deben evitar.

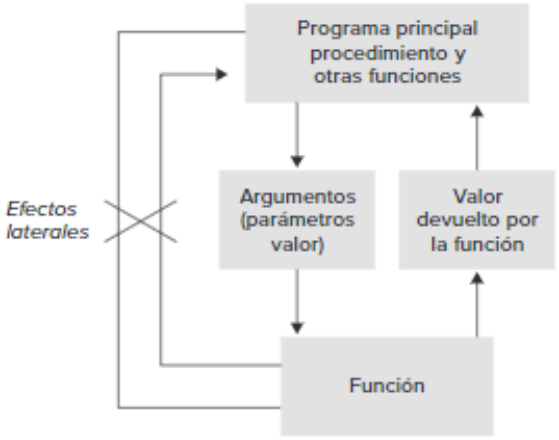


Figura 6.11. Efectos laterales en una función.

Los efectos laterales están considerados —normalmente— como una mala técnica de programación, pues hacen más difícil de entender los programas.

Toda la información que se transfiere entre procedimientos y funciones debe realizarse a través de la lista de parámetros y no a través de variables globales. Esto convertirá al procedimiento o función en módulos independientes que pueden ser comprobados y depurados por sí solos, lo que evitará preocuparnos por el resto de las partes del programa.

6.8. Recursión (recursividad)

Como ya se conoce, un subprograma puede llamar a cualquier otro subprograma y éste a otro, y así sucesivamente; dicho de otro modo, los subprogramas se pueden anidar. Se puede tener:

```
A llamar_a B, B llamar_a C, C llamar_a D
```

Cuando se produce el retorno de los subprogramas a la terminación de cada uno de ellos el proceso resultante será:

```
D retornar_a C, C retornar_a B, B retornar_a A
```

¿Qué sucedería si dos subprogramas de una secuencia son los mismos?

```
A llamar_a A
```

o bien:

```
A llamar_a B, B llamar_a A
```

En primera instancia, parece incorrecta. Sin embargo, existen lenguajes de programación –Pascal, C, entre otros– en que un subprograma puede llamarse a sí mismo.

Una función o procedimiento que se puede llamar a sí mismo se llama *recursivo*. La *recursión* (**recursividad**) es una herramienta muy potente en algunas aplicaciones, sobre todo de cálculo. La recursión puede ser utilizada como una alternativa a la repetición o estructura repetitiva. El uso de la recursión es particularmente idóneo para la solución de aquellos problemas que pueden definirse de modo natural en términos recursivos.

La escritura de un procedimiento o función recursiva es similar a sus homónimos no recursivos; sin embargo, para evitar que la recursión continúe indefinidamente es preciso incluir una condición de terminación.

La razón de que existan lenguajes que admiten la recursividad se debe a la existencia de estructuras específicas tipo *pilas* (*stack*, en inglés) para este tipo de procesos y memorias dinámicas. Las direcciones de retorno y el estado de cada subprograma se guardan en estructuras tipo pilas. En el capítulo 11 se profundizará en el tema de las pilas; por ahora nos centraremos sólo en el concepto de recursividad y en su comprensión con ejemplos básicos.

Krasner Podres 1095584-2202171939

EJEMPLO 6.13

Muchas funciones matemáticas se definen recursivamente. Un ejemplo de ello es el factorial de un número entero n.

La función factorial se define como

$$n! = \begin{cases} 1 \text{ si } n = 0 & 0! = 1 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \text{ si } n > 0 & n.\,(n-1)\,.\,(n-2)\ldots.3.2.1 \end{cases}$$

Si se observa la fórmula anterior cuando `n > 0`, es fácil definir `n!` en función de `(n-1)!` Por ejemplo, `5!`

```
5! = 5x4x3x2x1      = 120
4! = 4x3x2x1        = 24
3! = 3x2x1          = 6
2! = 2x1            = 2
1! = 1x1            = 1
0! = 1              = 1
```

Se pueden transformar las expresiones anteriores en

```
5! = 5x4!
4! = 4x3!
3! = 3x2!
2! = 2x1!
1! = 1x0!
```

En términos generales sería:

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ n(n-1)! & \text{si} \quad n > 0 \end{cases}$$

La función `FACTORIAL` de `N` expresada en términos recursivos sería:

```
FACTORIAL ← N * FACTORIAL(N - 1)
```

La definición de la función sería:

```
entero: función factorial(E entero: n)
//cálculo recursivo del factorial
inicio
   si n = 0 entonces
      devolver (1)
   si_no devolver (n * factorial(n - 1))
   fin_si
fin_función
```

Para demostrar cómo esta versión recursiva de `FACTORIAL` calcula el valor de $n!$, consideremos el caso de $n = 3$. Un proceso gráfico se representa en la figura 6.12.

Krasher Podres 1095564-2202171940

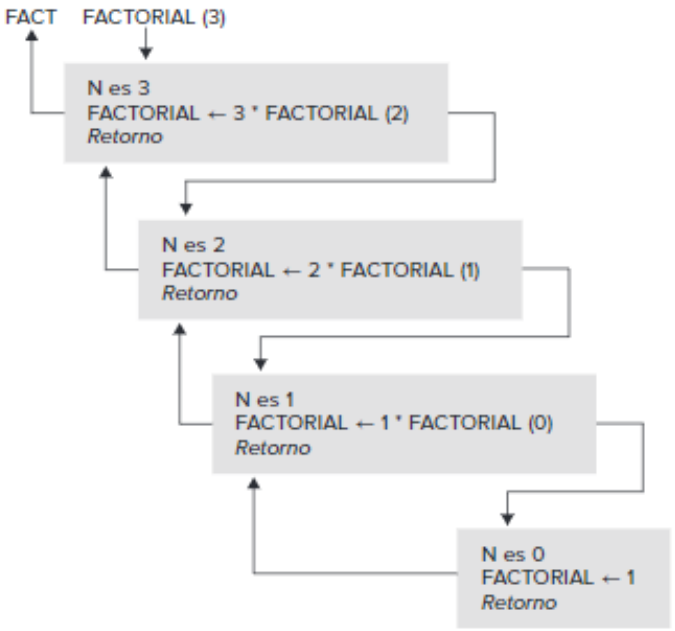


Figura 6.12. Cálculo recursivo de `FACTORIAL` de 3.

EJEMPLO 6.14

Otro ejemplo típico de una función recursiva es la serie Fibonacci. Esta serie fue concebida originalmente como modelo para el crecimiento de una granja de conejos (multiplicación de conejos) por el matemático italiano del siglo XVI, Fibonacci.

La serie es la siguiente:

```
1, 1, 2, 3, 5, 8, 13, 21, 34 ...
```

Esta serie crece muy rápidamente; como ejemplo, el término 15 es 610.

La serie de Fibonacci (*fib*) se expresa así:

$fib(1) = 1$
$fib(2) = 1$
$fib(n) = fib(n - 1) + fib(n - 2)$ para $n > 2$

Una función recursiva que calcula el elemento enésimo de la serie de Fibonacci es:

```
entero: función fibonacci(E entero: n)
//cálculo del elemento n-ésimo
inicio
   si (n = 1) o (n = 2) entonces
      devolver (1)
   si_no
      devolver (fibonacci(n - 2) + fibonacci(n - 1))
   fin_si
fin_función
```

Aunque es fácil de escribir la función de Fibonacci, no es muy eficaz definida de esta forma, ya que cada paso recursivo genera otras dos llamadas a la misma función.

6.9. Funciones en C/C++, Java y C#

La sintaxis básica y estructura de una función en C, C++, Java y C# son realmente idénticas.

```
valor_retorno nombre_función (tipo1 arg1, tipo2 arg2 …)
{  // equivalente a la palabra reservada en pseudocódigo inicio
   // cuerpo de la función
}  // equivalente a la palabra reservada en pseudocódigo fin
```

En Java, todas las funciones deben estar asociadas con alguna clase y se denominan **métodos**. En C++, las funciones asociadas con una clase se llaman **funciones miembro**. Las funciones C tradicionales y las no asociadas con ninguna clase en C++, se denominan simplemente **funciones no miembro**.

El nombre de la función y su lista de argumentos constituyen la ***signatura***. La lista de parámetros formales es la *interfaz de la función* con el mundo exterior, dado que es el punto de entrada para parámetros entrantes.

La descripción de una función se realiza en dos partes: *declaración de la función y definición de la función*. La **declaración de una función**, denominada también *prototipo de la función*, describe cómo se llama (invoca) a la función. Existen dos métodos para declarar una función:

1. Escribir la función completa antes de ser utilizada.
2. Definir el *prototipo de la función* que proporciona al compilador información suficiente para llamar a la función. El prototipo de una función es similar a la primera línea de la función, pero el prototipo no tiene cuerpo.

Prototipo de función
Definición función
cabecera
cuerpo

```
double precio_total (int número, double precio);
double precio_total (int número, double precio)
{
   const double IVA = 0.06; // impuesto 6%
   double subtotal;
      subtotal = número * precio;
   return (subtotal + IVA * subtotal);
}
```

Krasher Podres 1095564-2202171941

Paso de parámetros

El paso de parámetros varía ligeramente entre Java y C++. Desde el enfoque de Java:

- No existen punteros como en C y C++.
- Los tipos integrados o incorporados (*built-in*, denominados también tipos primitivos de datos) se pasan siempre por valor.
- Tipos objeto (similares a las clases que son parte de los paquetes estándar de Java) se pasan siempre por referencia.

Las variables de Java que representan tipos objeto se llaman *variables de referencia* y aquellas que representan tipos integrados se llaman *variables de no referencia.*

EJEMPLO 6.15

Función en C++

La función triángulo calcula el área de un triángulo en C++

```
// Función en C++
// Triángulo, cálculo del área o superficie
// Parámetros
// anchura - anchura del triángulo
// altura - altura del triángulo
// retorno (devuelve)
// Área del triángulo
float triángulo(float anchura, float altura)
{
   float área
   assert (anchura >= 0.0);
   assert (altura >= 0.0);
   return (área)
}
...
// Llamada por valor a la función área
Superficie = triángulo (2.5, 4.6); // paso de parámetros por valor
```

La llamada por valor ejecuta la función `triángulo`, calcula la fórmula del área del triángulo y su valor 11.50 se asigna a la variable `superficie`.

EJEMPLO 6.16

Función en Java

```
import java.awt.Point;     // se importa la clase
                           // Point parte del paquete
                           // estándar Java AWT
// paso por valor
public class DemoPasoParametros {
   public static void intercambio_por_valor (int x, int y){
      int aux;
      aux = x;
      x = y;
      Y = aunx;
   }
}
```

Krasher Padres 1095584-2202171942

```
public static void intercambio_por_referencia (Punto p)[ '
   int aux = p.x;
   p.x     = p.y;
   p.y     = aux;
   // …
]
// llamadas a la función
int m = 50;
int n = 75;
// …
intercambio_porvalor (m, n);
// …
punto unPunto= new Punto (-10, 50);
intercambio_porreferencia (unPunto)
// ...
```

En las líneas de código anteriores, `m` y `n` son variables integradas de tipo **`int`.** Para intercambiar los valores de las dos variables se pasan en el método `intercambio_porvalor`. Una copia de los valores `m` y `n` se pasan a la función `intercambio_porvalor` cuando se llama a la función.

En el caso de la llamada por referencia se ha instanciado e inicializado un objeto, `unPunto` (de la clase `Point` definido en el paquete `Java.awt.Point`) a los valores 50, 75. Cuando `unPunto` se pasa en un método llamado `intercambio_porreferencia`, el método intercambia el contenido de las coordenadas *x* y *y* del argumento. Es preciso observar que una variable referencia es realmente una dirección al objeto y no el objeto en sí mismo. Al pasar `unPunto`, en realidad se pasa la dirección del objeto y no una copia —como en el paso por valor— del objeto.

Esta característica de Java es equivalente semánticamente al modo en que funcionan las variables referencia y variables puntero en C++. En resumen, las variables de referencia en Java son muy similares a las referencias C++.

Kristher Padres 1095564-2202171947

6.10. Ámbito (alcance) y almacenamiento en C/C++ y Java

Cada identificador (nombre de una entidad) debe referirse a una única identidad (tal como una variable, función, tipo, etcétera). A pesar de este requisito, los nombres se pueden utilizar más de una vez en un programa. Un nombre puede reutilizarse mientras se utilice en diferentes contextos, a partir de los cuales los diferentes significados del nombre pueden ser empleados. El contexto utilizado para distinguir los significados de los nombres es su *alcance* o *ámbito* (*scope*). Un **ámbito** o **alcance** es una región del código de programa, donde se permite hacer referencia (uso) a un identificador.

> Un nombre (identificador) se puede referir a diversas entidades en diferentes ámbitos.

Los alcances están separados por los separadores `inicio-fin` (o llaves en los lenguajes C, C++, Java, etcétera). Los nombres son visibles desde su punto de declaración hasta el final del alcance en el que aparece la declaración.

Los identificadores definidos fuera de cualquier función tienen ámbito global, son accesibles desde cualquier parte del programa; sin embargo, los definidos en el cuerpo de una función tienen *ámbito local.*

La *clase de almacenamiento* de una variable puede ser *permanente* o *temporal.* Las variables globales son siempre permanentes, se crean e inicializan antes de que el programa arranque y permanecen hasta que se termina. Las variables temporales se asignan desde una sección de memoria llamada la pila (*stack*) en el principio del bloque.

Si se intenta asignar muchas variables temporales se puede obtener un error de desbordamiento de la pila. El espacio utilizado por las variables temporales se devuelve (se libera) a la pila al final del bloque. Cada vez que se entra al bloque, se inicializan las variables temporales.

> Las variables locales son temporales a menos que sean declaradas estáticas `static` en C++.

Definición y declaración de variables

El ámbito (alcance) de una variable es el área (bloque) del programa donde es válida la variable. En general, las definiciones o declaraciones de variables se pueden situar en cualquier parte de un programa donde esté permitida una sentencia. Una variable debe ser declarada o definida antes de que sea utilizada.

Regla

Es una buena idea definir un objeto cerca del punto en el cual se va a utilizar la primera vez.

Las variables pueden ser *globales* o *locales*. Una variable global es de alcance global y es válida desde el punto en que se declara hasta el final del programa. Su duración es la del programa, hasta que se acaba su ejecución. Una variable local es aquella que está definida en el interior del cuerpo de una función y es accesible sólo dentro de dicha función. El ámbito de una variable local se limita al bloque donde está declarada y no puede ser accedida (leída o asignada un valor) fuera de ese bloque.

En el cuerpo o bloque de una función se pueden definir variables locales que son "locales" a dicha función y sus nombres sólo son visibles en el ámbito de la función. Las variables locales sólo existen mientras la función se está ejecutando.

Krasner Podres 1095564-2202171949

Un bloque es una sección de código encerrada entre inicio y fin (en el caso de C/C++ o Java/C#, encerrado entre llaves, { }).

Los nombres de las variables locales a una función son visibles sólo en el ámbito de la función y existen sólo mientras la función se está ejecutando. La ejecución se termina cuando se encuentra una sentencia `devolver (return)` y produce como resultado el valor especificado en dicha sentencia. Es posible declarar una variable local con el mismo nombre que una variable global, pero en el bloque donde está definida la variable local tiene prioridad sobre la variable global y se dice que esta variable se encuentra *oculta*. *Si una variable local oculta a una global* para que tenga el mismo nombre ***entonces***, se dice que la variable global no es posible.

Una variable global es aquella que se define fuera del cuerpo de las funciones y están disponibles en todas las partes del programa, incluso en otros archivos como en lenguajes C++ donde un programa puede estar en dos archivos. En el caso de que un programa esté compuesto por dos archivos, en el primero se define la variable global y se declara en el segundo archivo donde se puede utilizar.

EJEMPLO 6.17

Variables locales y globales en C++

alcance

```
int cuenta;           // variable global
int main ( )          // función principal
{
   int local:         // variable local
      cuenta = 100:
   local = 500;
```

global local
alcance
local_uno

```
    {
       int local_uno;
       local_uno = cuenta + local;
    }
    // no se puede utilizar local_uno
}
```

Si en el segmento de código siguiente se declara una nueva variable local `cuenta`, ésta se oculta a la variable global `cuenta` inicializada a 100 en el cuerpo de la función.

```
int total;
int cuenta;

int main( )
{
   total = 0;
   cuenta = 100;

   int cuenta;
   cuenta = 0;
   while (true) {
      if (cuenta > 10)
         break;
      total += cuenta;
      ++cuenta;
      }
   }
   ++cuenta;
   return (0);
}
```

Krasher Padres 1095584-22021171950

6.11. Sobrecarga de funciones en C++ y Java

Algunos lenguajes de programación como C++ o Java permiten la sobrecarga de funciones (funciones miembro en C++, métodos en Java). La **sobrecarga de funciones**, que aparecen en el mismo ámbito, significa que se pueden definir múltiples funciones con el mismo nombre, pero con listas de parámetros diferentes.

Sobrecarga de una función es usar el mismo nombre para diferentes funciones, distintas unas de otras por sus listas de parámetros.

En realidad, ésta es una propiedad que facilita la tarea al programador cuando se desea diseñar funciones que realizan la misma tarea general pero que se aplican a tipos de parámetros diferentes. Estas funciones se pueden llamar sin preocuparse sobre cuál se invoca, ya que el compilador detecta el tipo de dato de los parámetros y ejecuta la función asociada a ellos.

Por ejemplo, se trata de ejecutar una función que imprima los valores de determinadas variables de diferentes tipos de datos: `car`, `entero`, `real`, `lógico`, etcétera. Así algunas funciones que realizan estas tareas serían:

```
nada ImprimirEnteros (entero n)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', n)
fin
nada ImprimirCar(car c)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', c)
fin

nada ImprimirReal(real r)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', r)
fin

nada ImprimirLógico (lógico l)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', l)
fin
```

Krasher Padres 1095564-2202171951

Se necesitan cuatro funciones diferentes con cuatro nombres distintos; si se utilizan funciones sobrecargadas, en lugar de utilizar un nombre para cada tipo de impresión de datos, se puede utilizar una función sobrecargada con el mismo nombre `Imprimir` y con distintos parámetros.

1095564-2202171951

```
nada Imprimir (entero n)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', n)
fin

nada Imprimir (car c)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', c)
fin

nada Imprimir (real r)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', r)
fin

nada Imprimir (lógico l)
inicio
   escribir ('Visualizar')
   escribirn('El valor es', l)
fin
```

El código que llama a estas funciones podría ser:

```
Imprimir (entero1)
Imprimir (car1)
Imprimir (real1)
Imprimir (lógico1)
```

De este modo, el nombre de la función tiene cuatro definiciones diferentes ya que el nombre `Imprimir` está sobrecargado.

La sobrecarga es una característica muy notable ya que hace a un programa más fácil de leer. Cuando se invoca a una función sobrecargada el compilador comprueba el número y tipo de argumentos en dicha llamada.

EJERCICIO

Sobrecarga de la función media (media aritmética de dos o tres números reales).

```
real media (real n1, real n2)
inicio
   devolver ((n1 + n2)/ 2.0)
fin
real media (real n1, real n2, real n3) inicio
   devolver ((n1 + n2 + n3)/ 3.0)
fin
```

Algunas llamadas a la función son:

```
media (4.5, 7.5)
media (3.5, 5.5, 10.5)
```

Krasner Padres 1095564-2202171952

1095564220217 1952

C , Pascal y Fortran no soportan sobrecarga de funciones. *C++ y Java soportan sobrecarga de funciones miembro y métodos.*

EJEMPLO

Función cuadrado que eleva al cuadrado el valor del argumento.

```
entero cuadrado (entero n)
inicio
   devolver (n*n)
fin
```

```
entero cuadrado (real n)
inicio
   devolver (n * n)
fin
```

Sobrecarga en C++

Las funciones anteriores escritas en C++

```
int cuadrado (int n)
{
   return (n * n);
}
```

```
float cuadrado (float n)
{
   return (n * n);
}
```

Sobrecarga en Java

En Java se pueden diseñar en una clase métodos con el mismo nombre, e incluso en la biblioteca de clases de Java, la misma situación. Dos características diferencian los métodos con igual nombre:

- El número de argumentos que aceptan.
- El tipo de dato u objetos de cada argumento.

Estas dos características constituyen la *signatura* de un método. El uso de varios métodos con el mismo nombre y signaturas diferentes se denomina *sobrecarga;* ésta puede eliminar la necesidad de escribir métodos diferentes que realizan la misma acción. La sobrecarga facilita que existan métodos que se comportan de modo diferente basándose en los argumentos que reciben.

Cuando se llama a un método de un objeto, en Java, hace corresponder el nombre del método y los argumentos para seleccionar cuál es la definición a ejecutar.

Para crear un método sobrecargado, se crean diferentes definiciones de métodos en una clase, cada uno con el mismo nombre, pero diferente lista de argumentos. La diferencia puede ser el número, el tipo de argumentos o ambos. Java permite la sobrecarga de métodos pero cada lista de argumentos es única para el mismo nombre del método.

Krasner Podres 1095584-2202172007

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

1095584220217200

6.1. *Realización del factorial de un número entero.*

```
entero: función factorial(E entero: n)
var entero: f, i
inicio
   si n = 0 entonces
      devolver (1)
   si_no
      desde i ← 1 hasta n hacer
         f ← f*i
      fin_desde
      devolver (f)
   fin_si
fin_función
```

6.2. *Diseñar un algoritmo que calcule el máximo común divisor de dos números mediante el algoritmo de Euclides.*

Sean los dos números `A` y `B`. El método para hallar el máximo común divisor (`mcd`) de dos números `A` y `B` por el método de Euclides es:

1. Dividir el número mayor (`A`) por el menor (`B`). Si el resto de la división es cero, el número `B` es el máximo común divisor.
2. Si la división no es exacta, se divide el número menor (`B`) por el resto de la división anterior.
3. Se siguen los pasos anteriores hasta obtener un resto de valor cero. El último divisor es el `mcd` buscado.

Algoritmo

```
entero función mcd(E entero: a, b)
inicio
   mientras a <> b hacer
      si a > b entonces
         a ← a - b
```

```
      si_no
         b ← b - a
      fin_si
   fin_mientras
   devolver(a)
fin_función
```

6.3. *Para calcular el máximo común divisor (mcd) de dos números se recurre a una función específica definida con un subprograma. Se desea calcular la salida del programa principal con dos números A y B, cuyos valores son 10 y 25, es decir, el mcd (A, B) y comprobar el método de paso de parámetros por valor.*

```
algoritmo maxcomdiv
var
   entero: N, X, Y
inicio //programa principal
   x ← 10
   y ← 25
   n ← mcd(x, y)
   escribir(x, y, n)
fin
entero función mcd(E entero: a, b)
inicio
   mientras a <> b hacer
      si a > b entonces
      a ← a - b
      si_no
         b ← b - a
      fin_si
   fin_mientras
   devolver (a)
fin_función
```

Los parámetros formales son a y b y recibirán los valores de x y y.

```
a = 10
b = 25
```

Las variables locales a la función son A y B y no modificarán los valores de las variables x y y del algoritmo principal.

Variables del programa principal		*Variables de la función*	
x	y	a	b
10	25	10	25

Las operaciones del algoritmo son:

```
a = 10     b = 25
```

1. b > a realizará la operación b ← b - a
   y por consiguiente *b* tomará el valor 25 - 10 = 15
   y a sigue valiendo 10
2. a = 10 b = 15
   se realiza la misma operación anterior
   b ← b - a, es decir, b = 5
   a permanece inalterable

Krasher Padres 1095564-2202172012

3. a = 10 b = 5

   como a > b entonces se realiza a ← a - b, es decir, a = 5

Por lo tanto, los valores finales serían:

a = 5 b = 5 mcd(a, b) = 5

Como los valores a y b no se pasan al algoritmo principal, el resultado de su ejecución será:

10 25 5

6.4. *Realizar un algoritmo que permita ordenar tres números mediante un procedimiento de intercambio en dos variables (paso de parámetros por referencia).*

El algoritmo que permite realizar el intercambio de los valores de variables numéricas es:

```
AUXI ← A
A    ← B
B    ← AUXI
```

y la definición del procedimiento será:

```
PROCEDIMIENTO intercambio (E/S real: a, b)
var real : auxi

inicio
   auxi ← a
   a    ← b
   b    ← auxi
fin_procedimiento
```

El algoritmo de ordenación se realizará mediante llamadas al procedimiento *intercambio*.

```
algoritmo Ordenar_3_números
var real : x,y,z

inicio
   escribir('Deme 3 números reales')
   leer(x, y, z)
   si x > y entonces
      intercambio (x, y)
   fin_si
   si y > z entonces
      intercambio (y, z)
   fin_si
   si x > y entonces
      intercambio (x, y)
   fin_si
   escribir(x, y, z)
fin
```

Paso de parámetros por referencia

Los tres números X, Y, Z que se van a ordenar son:

132 45 15

Los pasos sucesivos al ejecutarse el algoritmo o programa principal son:

1. Lectura X, Y, Z parámetros actuales:

Krasher Padres 1095564-2202172012

X = 132
Y = 45
Z = 15

2. Primera llamada al procedimiento `intercambio(a, b)` `x > y`. La correspondencia entre parámetros será la siguiente:

parámetros actuales *parámetros formales*

X ⟶ A
y ⟶ B

Al ejecutarse el procedimiento se intercambiarán los valores de A y B que se devolverán a las variables X y Y; luego valdrán:

X = 45
Y = 132

3. Segunda llamada al procedimiento *intercambio* con Y > Z (ya que Y = 132 y Z = 15)

parámetros actuales *parámetros formales*

Y ⟶ A
Z ⟶ B

Antes llamada al procedimiento Y = 132, Z = 15.

Después terminación del procedimiento Z = 132, Y = 15, ya que A y B han intercambiado los valores recibidos, 132 y 15.

4. Los valores actuales de X, Y, Z son 45, 15, 132; por consiguiente, X > Y y habrá que hacer otra nueva llamada al procedimiento *intercambio*.

parámetros actuales *parámetros formales*

X(45) ⟶ A(45)
Y(15) ⟶ B(15)

Después de la ejecución del procedimiento A y B intercambiarán sus valores y valdrán A = 15, B = 45, por lo que se pasan al algoritmo principal X = 15, Y = 45. Por consiguiente, el valor final de las tres variables será:

X = 15 Y = 45 Z = 132

ya ordenados de modo creciente.

6.5. *Diseñar un algoritmo que llame a la función* **signo(X)** *y calcule: a) el signo de un número, b) el signo de la función coseno.*

Variables de entrada: P (real)

Variables de salida: Y-signo del valor P-(entero) Z-signo del coseno de P-(entero);

Pseudocódigo

```
algoritmo signos
var entero: y, z
   real: P
inicio
   leer(P)
   Y ← signo(p)
   Z ← signo(cos (p))
   escribir(Y, Z)
fin
```

Krasher Padres 1095584-2202172013

```
entero función signo(E real: x)
   inicio
      si x > 0 entonces
         devolver (1)
      si_no
         si x < 0 entonces
            devolver (-1)
         si_no
            devolver (0)
         fin_si
      fin_si
fin_función
```

Notas de ejecución

Parámetro actual	*Parámetro formal*
P	x

El parámetro formal x se sustituye por el parámetro actual. Así, por ejemplo, si el parámetro P vale -1.45. Los valores devueltos por la función Signo que se asignará a las variables Y, Z son:

```
Y ← Signo(-1.45)
Z ← Signo(Cos (-1.45))
```

resultando:

```
Y = -1
Z = 1
```

Krasner Padres 1095564-2202172014

1095564220217201

CONCEPTOS CLAVE

- alcance
- ámbito
- ámbito global
- ámbito local
- argumento
- argumento actual
- argumentos formales
- argumentos reales
- biblioteca estándar
- cabecera de función
- clase de almacenamiento
- cuerpo de la función
- función
- función invocada
- función llamada
- función llamadora
- función recursiva
- módulo
- parámetro
- parámetro actual
- parámetros formales
- parámetros reales
- paso por referencia
- paso por valor
- procedimiento
- prototipo de función
- sentencia `devolver (return)`
- subprograma
- rango
- variable global
- variable local

RESUMEN

Aunque los conceptos son similares, las unidades de programas definidas por el usuario se conocen generalmente por el término de *subprogramas* para representar los módulos correspondientes; sin embargo, se denominan con nombres diferentes en los distintos lenguajes de programación. Así en los lenguajes **C** y **C++** los subprogramas se denominan *funciones*; en los lenguajes de programación orientada a objetos (**C++**, **Java** y **C♯**) y siempre que se definen dentro de las clases, se les suele también denominar *métodos* o funcio-

nes miembro; en **Pascal**, son *procedimientos* y *funciones;* en **Módula-2** los nombres son PROCEDIMIENTOS (procedures, incluso aunque algunos de ellos son realmente funciones); en **Cobol** se conocen como *párrafos* y en los "viejos" **Fortran** y **Basic** se les conoce como *subrutinas* y *funciones*. Los conceptos más importantes sobre funciones y procedimientos son los siguientes:

1. Las funciones y procedimientos se pueden utilizar para romper un programa en módulos de menor complejidad. De esta forma un trabajo complejo se puede descomponer en otras unidades más pequeñas que interactúan unas con otras de un modo controlado. Estos módulos tienen las siguientes propiedades:
   a) El propósito de cada función o procedimiento debe estar claro y ser simple.
   b) Una función o procedimiento debe ser lo bastante corta como para ser comprendida en toda su entidad.
   c) Todas sus acciones deben estar interconectadas y trabajar al mismo nivel de detalle.
   d) El tamaño y la complejidad de un subprograma se pueden reducir llamando a otros subprogramas para que hagan subtareas.
2. Las funciones definidas por el usuario son subrutinas que realizan una operación y devuelven un valor al entorno o módulo que le llamó. Los argumentos pasados a las funciones se manipulan por la rutina para producir un valor de retorno. Algunas funciones calculan y devuelven valores, otras funciones no. Una función que no devuelve ningún valor, se denomina función `void` en el caso del lenguaje C.
3. Los procedimientos no devuelven ningún valor al módulo que le invocó. En realidad, los procedimientos ya se conservan sólo en algunos lenguajes procedimentales como Pascal. En el resto de los lenguajes sólo se implementan funciones y los procedimientos son equivalentes a funciones que no devuelven valor.
4. Una llamada a una función que devuelve un valor, se encuentra normalmente en una sentencia de asignación, una expresión o una sentencia de salida.
5. Los componentes básicos de una función son la cabecera de la función y el cuerpo de la función.
6. Los argumentos son el medio por el cual un programa llamador comunica o envía los datos a una función. Los parámetros son el medio por el cual una función recibe los datos enviados o comunicados. Cuando una función se llama, los argumentos reales en la llamada a la función se pasan a dicha función y sus valores se sustituyen en los parámetros formales de la misma.
7. Después de pasar los valores de los parámetros, el control se pasa a la función. El cálculo comienza en la parte superior de la función y prosigue hasta que se termina, en cuyo momento el resultado se devuelve al programa llamador.
8. Cada variable utilizada en un programa tiene un ámbito (rango o alcance) que determina en qué parte del programa se puede utilizar. El ámbito de una variable es local o global y se determina por la posición donde se sitúa la variable. Una variable local se define dentro de una función y sólo se puede utilizar dentro de la definición de dicha función o bloque. Una variable global está definida fuera de una función y se puede utilizar en cualquier función a continuación de la definición de la variable. Todas las variables globales que no son inicializadas por el usuario, normalmente se inicializan a cero por la computadora.
9. Una solución recursiva es una en que la solución se puede expresar en términos de una versión más simple de sí misma. Es decir, una función recursiva se puede llamar a sí misma.
10. Si una solución de un problema se puede expresar repetitivamente o recursivamente con igual facilidad, la solución repetitiva es preferible, ya que se ejecuta más rápidamente y utiliza menos memoria. Sin embargo, en muchas aplicaciones avanzadas la recursión es más simple de visualizar y el único medio práctico de implementar una solución.

EJERCICIOS

6.1. Diseñar una función que calcule la media de tres números leídos del teclado y poner un ejemplo de su aplicación.

6.2. Realizar la función `FACTORIAL` que calcule el factorial de un número entero en el rango 100 a 1 000 000.

6.3. Escribir un algoritmo para calcular el máximo común divisor de cuatro números basado en un subalgoritmo función mcd (máximo común divisor de dos números).

6.4. Crear una función que encuentre el mayor de dos números enteros.

Krasner Podres 1095584-2202172015

6.5. Diseñar una función que calcule x^n para `x`, variable real y `n` variable entera.

6.6. Delinear un procedimiento que acepte un número de mes, un número de día y un número de año y los visualice en el formato:

```
dd/mm/aa
```

Por ejemplo, los valores 19, 09, 1987 se visualizarían como:

```
19/9/87
```

y para los valores 3, 9 y 1905

```
3/9/05
```

6.7. Realizar un procedimiento que realice la conversión de coordenadas polares (r, θ) a coordenadas cartesianas (x, y).

$x = r.cos\ (\theta)$
$y = r.sen\ (\theta)$

6.8. Escribir una función `Salario` que calcule los salarios de un trabajador para un número dado de horas trabajadas y un salario hora. Las horas que superen las 40 horas semanales se pagarán como extras con un salario hora 1.5 veces el salario ordinario.

6.9. Crear una función booleana `Dígito` que determine si un carácter es uno de los dígitos 0 al 9.

6.10. Diseñar una función que permita devolver el valor absoluto de un número.

6.11. Realizar un procedimiento que obtenga la división entera y el resto de la misma utilizando únicamente los operadores suma y resta.

6.12. Escribir una función que permita deducir si una fecha leída del teclado es válida.

6.13. Diseñar un algoritmo que transforme un número introducido por teclado en notación decimal a notación romana. El número será entero positivo y no excederá de 3.000.

6.14. Escribir el algoritmo de una función recursiva que: *a*) calcule el factorial de un número entero positivo, *b*) la potencia de un número entero positivo.

Krasher Podres 1095564-2202172016

PARTE II

ESTRUCTURA DE DATOS

Krasher Padres 1095564-2202172018

1095564220217201B

CONTENIDO

CAPÍTULO 7 Estructuras de datos I (*arrays* –arreglos– y estructuras)

CAPÍTULO 8 Las cadenas de caracteres

CAPÍTULO 9 Archivos (ficheros)

CAPÍTULO 10 Ordenación, búsqueda e intercalación

CAPÍTULO 11 Ordenación, búsqueda y fusión externa (archivos)

CAPÍTULO 12 Estructuras dinámicas lineales de datos (pilas, colas y listas enlazadas)

CAPÍTULO 13 Estructuras de datos no lineales (árboles y grafos)

CAPÍTULO 14 Recursividad

1095564220217 2019

CAPÍTULO 7

Estructuras de datos I (*arrays* –arreglos– y estructuras)[1]

7.1. Introducción a las estructuras de datos
7.2. Arrays (arreglos) unidimensionales: los vectores
7.3. Operaciones con vectores
7.4. Arrays (arreglos) de varias dimensiones
7.5. Arrays (arreglos) multidimensionales
7.6. Almacenamiento de arrays en memoria
7.7. Estructuras *versus* registros
7.8. Arrays (arreglos) de estructuras
7.9. Uniones
7.10. Enumeraciones
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Krasher Padres 1095564-2202172020

1095564-2202172020

INTRODUCCIÓN

En los capítulos anteriores se ha introducido el concepto de datos de tipo simple que representan valores de tipo simple, como un número entero, real o un carácter. Sin embargo, en muchas situaciones se necesita procesar una colección de valores que están relacionados entre sí por algún método, por ejemplo, una lista de calificaciones, una serie de temperaturas medidas a lo largo de un mes, etcétera. El procesamiento de tales conjuntos de datos, utilizando datos simples, puede ser extremadamente difícil y por ello la mayoría de los lenguajes de programación incluyen características de estructuras de datos. *Las estructuras de datos básicas* que soportan la mayoría de los lenguajes de programación son los *arrays* –concepto matemático de *vector* y *matriz*–.

Un **array**, o **arreglo** (*en Latinoamérica*), es una secuencia de posiciones de la memoria central a las que se puede acceder directamente, que contiene datos del mismo tipo y pueden ser seleccionados individualmente mediante el uso de subíndices. Este capítulo estudia el concepto de *arrays unidimensionales y multidimensionales*, así como el procesamiento de los mismos.

[1] El término *array* se conserva en inglés por su amplia aceptación en la comunidad de ingeniería informática y de sistemas; sin embargo, es preciso constatar que en prácticamente toda Latinoamérica (al menos en muchos de los numerosos países que conocemos y con los que tenemos relaciones académicas y personales) el término empleado como traducción es **arreglo**. Diccionario de la Real Academia Española (Edición 23, Madrid 2014 y la última versión electrónica 23.3) no considera ninguno de los dos términos como válidos, aunque la acepción 2 de la definición de **arreglo** pudiera ser ilustrativa del porqué de la adopción del término por la comunidad latinoamericana: **"Regla, orden, coordinación"**.

7.1. Introducción a las estructuras de datos

Una *estructura de datos* es una colección de datos que puede ser caracterizada por su organización y las operaciones que se definen en ella.

Las estructuras de datos son muy importantes en los sistemas de computadora; los tipos de datos más frecuentes utilizados en los diferentes lenguajes de programación son:

Datos simples	*estándar*	entero (`integer`) real (`real`) carácter (`char`) lógico (`boolean`)
	definido por el programador (no estándar)	subrango (`subrange`) enumerativo (`enumerated`)
Datos estructurados	*estáticos*	arrays (`vectores/matrices`) registros (`record`) ficheros (`archivos`) conjuntos (`set`) cadenas (`string`)
	dinámicos	listas (`pilas/colas`) listas enlazadas árboles grafos

Los tipos de datos *simples* o *primitivos* significan que no están compuestos de otras estructuras de datos; los más frecuentes y utilizados por casi todos los lenguajes son: *enteros, reales* y *carácter (char)*, siendo los tipos *lógicos, subrango* y *enumerativos* propios de lenguajes estructurados como Pascal. Los tipos de datos compuestos están construidos basados en tipos de datos primitivos; el ejemplo más representativo es la *cadena (string)* de caracteres.

Los tipos de datos simples pueden ser organizados en diferentes estructuras de datos: *estáticas* y *dinámicas*. Las **estructuras de datos estáticas** son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa. Estas estructuras están implementadas en casi todos los lenguajes: *array* (vectores/tablas-matrices), *registros, ficheros* o *archivos* (los *conjuntos* son específicos del lenguaje Pascal). Las **estructuras de datos dinámicas** no tienen las limitaciones o restricciones en el tamaño de memoria ocupada que son propias de las estructuras estáticas. Mediante el uso de un tipo de datos específico, denominado *puntero*, es posible construir estructuras de datos dinámicas que son soportadas por la mayoría de los lenguajes que ofrecen soluciones eficaces y efectivas en la solución de problemas complejos –Pascal es el lenguaje tipo por excelencia con posibilidad de estructuras de datos dinámicos–. Las estructuras dinámicas por excelencia son las *listas* –enlazadas, pilas, colas–, *árboles* –binarios, árbol-b, búsqueda binaria– y *grafos*.

La elección del tipo de estructura de datos idónea a cada aplicación dependerá esencialmente del tipo de aplicación y, en menor medida, del lenguaje, ya que en aquellos en que no está implementada una estructura –por ejemplo, las listas y árboles no los soporta Cobol– deberá ser simulada con el algoritmo adecuado, dependiendo del propio algoritmo y de las características del lenguaje su fácil o difícil solución.

Una característica importante que diferencia a los tipos de datos es la siguiente: los tipos de datos simples tienen como característica común que cada variable representa a un elemento; los tipos de datos estructurados tienen como característica común que un *identificador* (nombre) puede representar múltiples datos individuales, pudiendo cada uno de éstos ser referenciado independientemente.

Krasher Podres 1095584-2202172021

7.2. Arrays (arreglos) unidimensionales: los vectores

Un *array* o *arreglo (matriz* o *vector)* es un conjunto finito y ordenado de elementos homogéneos. La propiedad *ordenado* significa que el elemento primero, segundo, tercero, ..., enésimo de un *array* puede ser identificado. Los elementos de un *array* son homogéneos, es decir, del mismo tipo de datos. Un *array* puede estar compuesto de todos sus elementos de tipo cadena, otro puede tener todos sus elementos de tipo entero, etcétera. Los *arrays* se conocen también como *matrices* –en matemáticas– y *tablas* –en cálculos financieros–.

El tipo más simple de *array* es el *array unidimensional o vector* (matriz de una dimensión). Un vector de una dimensión denominado `NOTAS` que consta de *n* elementos se puede representar por la figura 7.1.

NOTAS(1)	NOTAS(2)	.....	NOTAS(I)	.....	NOTAS(N)

Figura 7.1. Vector.

El *subíndice o índice* de un elemento (1, 2, ..., *i*, *n*) designa su posición en la ordenación del vector. Otras posibles notaciones del vector son:

```
a1, a2, ..., ai, ..., an          en matemáticas y algunos lenguajes (VB 6.0 y VB.Net)
A(1), A(2), ..., A(i), ..., A(n)
A[1], A[2], ..., A[i], ..., A[n]   en programación (Pascal y C)
```

Observe que sólo el vector global tiene nombre `(NOTAS)`. Los elementos del vector se referencian por su *subíndice* o índice (*subscript*), es decir, su posición relativa en el valor.

En algunos libros y tratados de programación, además de las notaciones anteriores, suele utilizarse esta otra:

```
A(L:U) = [A(I)]
para I = L, L+1, ..., U-1, U      donde cada elemento A(I) es de tipo de datos T
```

que significa: `A`, vector unidimensional con elementos de datos tipo `T`, cuyos subíndices varían en el rango de `L` a `U`, lo cual significa que el índice no tiene por que comenzar necesariamente en 0 o en 1.

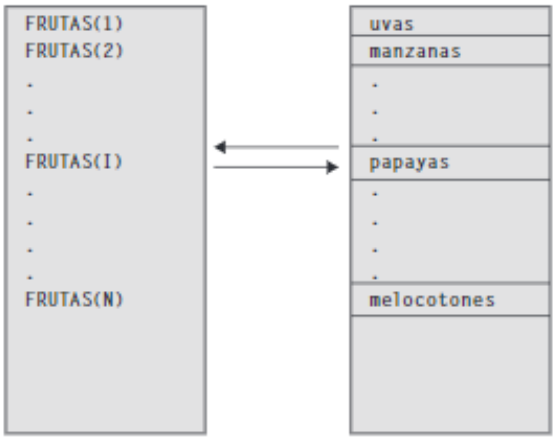
Como ejemplo de un vector o *array* unidimensional, se puede considerar el vector `TEMPERATURA` que contiene las temperaturas horarias registradas en una ciudad durante las veinticuatro horas del día. Este vector constará de veinticuatro elementos de tipo real, ya que las temperaturas normalmente no serán enteras siempre.

El valor mínimo permitido de un vector se denomina *límite inferior* del vector (`L`) y el valor máximo permitido se denomina *límite superior* (`U`). En el ejemplo del vector `TEMPERATURAS` el límite inferior es 1 y el superior 24.

```
TEMPERATURAS(I)          donde 1 <= I <= 24
```

El número de elementos de un vector se denomina *rango de vector*. El rango del vector `A(L:U)` es `U-L+1`. El rango del vector `B(1:n)` es `n`.

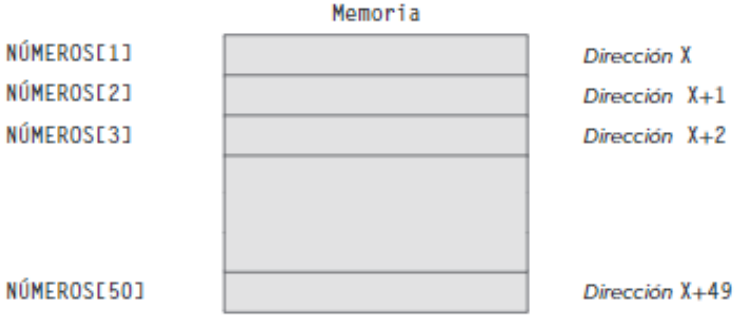
Los vectores, como ya se ha comentado, pueden contener datos no numéricos, es decir, tipo "carácter". Por ejemplo, un vector que representa las frutas que se venden en un supermercado:



Krasher Padres 1095564-2202172026

Otro ejemplo de un vector pueden ser los nombres de los alumnos de una clase. El vector se denomina ALUMNOS y tiene treinta elementos de rango.

Los vectores se almacenan en la memoria central de la computadora en un orden adyacente. Así, un vector de cincuenta números denominado NÚMEROS se representa gráficamente por cincuenta posiciones de memoria sucesivas.



Cada elemento de un vector se puede procesar como si fuese una variable simple al ocupar una posición de memoria. Así:

```
NÚMEROS[25] ← 72
```

almacena el valor entero o real 72 en la posición 25ª del vector NÚMEROS y la instrucción de salida:

```
escribir (NÚMEROS[25])
```

visualiza el valor almacenado en la posición 25ª, en este caso 72.

Esta propiedad significa que cada elemento de un vector –y posteriormente una tabla o matriz– es accesible directamente y es una de las *ventajas* más importantes de usar un vector: *almacenar un conjunto de datos*.

Consideremos un vector X de ocho elementos.

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
14.0	12.0	8.0	7.0	6.41	5.23	6.15	7.25
Elemento 1º	Elemento 2º						Elemento 8º

Algunas instrucciones que manipulan este vector se representan en la tabla 7.1.

Tabla 7.1. Operaciones básicas con vectores.

Acciones	Resultados
`escribir(X[1])`	Visualiza el valor de X[1] o 14.0.
`X[4] ← 45`	Almacena el valor 45 en X[4].
`SUMA ← X[1]+X[3]`	Almacena la suma de X[1] y x[3] o bien 22.0 en la variable SUMA.
`SUMA ← SUMA+X[4]`	Añade en la variable SUMA el valor de X[4], es decir, SUMA = 67.0.
`X[5] ← x[5]+3.5`	Suma 3.5 a X[5]; el nuevo valor de X[5] será 9.91.
`X[6] ← X[1]+X[2]`	Almacena la suma de X[1] y X[2] en X[6]; el nuevo valor de X[6] será 26.5.

Krasner Podres 1095584-2202172028

Antes de pasar a tratar las diversas operaciones que se pueden efectuar con vectores, consideremos la notación de los diferentes elementos.

Supongamos un vector V de ocho elementos.

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]
12	5	-7	14.5	20	1.5	2.5	-10

Los subíndices de un vector pueden ser enteros, variables o expresiones enteras. Así, por ejemplo, si

```
I ← 4
V[I+1]      representa el elemento V(5) de valor 20
V[I+2]      representa el elemento V(6) de valor 1.5
V[I-2]      representa el elemento V(2) de valor 5
V[I+3]      representa el elemento V(7) de valor 2.5
```

Los *arrays* unidimensionales (posteriormente se verán los *arrays* multidimensionales), necesitan ser dimensionados previamente a su uso dentro de un programa.

7.3. Operaciones con vectores

Un vector, como ya se ha mencionado, es una secuencia ordenada de elementos como:

```
X[1], X[2], ..., X[n]
```

El límite inferior no tiene por qué empezar en uno. El vector L:

```
L[0], L[1], L[2], L[3], L[4], L[5]
```

contiene seis elementos, en el que el primer elemento comienza en cero. El vector P, cuyo rango es 7 y sus límites inferior y superior son –3 y 3, es:

```
P[-3], P[-2], P[-1], P[0], P[1], P[2], P[3]
```

Las operaciones que se pueden realizar con vectores durante el proceso de resolución de un problema son:

- *Asignación.*
- *Lectura/escritura.*
- *Recorrido* (acceso secuencial).
- *Actualizar* (añadir, borrar, insertar).
- *Ordenación.*
- *Búsqueda.*

En general, las operaciones con vectores implican el procesamiento o tratamiento de los elementos individuales del vector.

Las notaciones algorítmicas que utilizaremos en este libro son:

```
tipo
array [lfminf .. lfmsup] de tipo : nombre_array
```

nombre_array	nombre válido del *array*
lfminf..lfmsup	límites inferior y superior del rango del *array*
tipo	tipo de datos de los elementos del *array*: entero, real, carácter

```
tipo
   array[1..10] de carácter : NOMBRES
var
   NOMBRES : N
```

Krasher Podres 1095584-2202172029

significa que NOMBRES es un *array* (vector) unidimensional de diez elementos (1 a 10) de tipo carácter.

```
tipo
   array['A'..'Z'] de real : LISTA
var
   LISTA : L
```

representa un vector cuyos subíndices son A, B, ... y cuyos elementos son de tipo real.

```
tipo
   array[0..100] de entero : NÚMERO
var
   NÚMERO:NU
```

NÚMERO es un vector cuyos subíndices van de 0 a 100 y de tipo entero.

Las operaciones que analizaremos en esta sección serán: *asignación, lectura/escritura, recorrido* y *actualización*, dejando por su especial relevancia como tema exclusivo de un capítulo la *ordenación* o *clasificación* y *búsqueda*.

7.3.1. Asignación

La asignación de valores a un elemento del vector se realizará con la instrucción de asignación:

```
A[29] ← 5          asigna el valor 5 al elemento 20 del vector A
```

Si se desea asignar valores a todos los elementos de un vector, se debe recurrir a estructuras repetitivas (**desde, mientras** o **repetir**) e incluso selectivas (**si-entonces, según**).

```
leer(A[i])
```

Si se introducen los valores 5, 7, 8, 14 y 12 mediante asignaciones:

```
A[1] ← 5
A[2] ← 7
A[3] ← 8
A[4] ← 14
A[5] ← 12
```

El ejemplo anterior ha asignado diferentes valores a cada elemento del vector A; si se desea dar el mismo valor a todos los elementos, la notación algorítmica se simplifica con el formato:

```
desde i = 1 hasta 5 hacer
   A[i] ←8
fin_desde
```

donde A[i] tomará los valores numéricos:

```
A[1] = 8, A[2] = 8, ..., A[5] = 8
```

Se puede utilizar también la notación:

```
A ←8
```

para indicar la asignación de un mismo valor a cada elemento de un vector A. Esta notación se considerará con mucho cuidado para evitar confusión con posibles variables simples numéricas de igual nombre (A).

7.3.2. Lectura/escritura de datos

La lectura/escritura de datos en *arrays* u operaciones de entrada/salida normalmente se realizan con estructuras repetitivas, aunque puede también hacerse con estructuras selectivas. Las instrucciones simples de lectura/escritura se representarán como:

```
leer(V[5])          leer el elemento V[5] del vector V
```

Krasher Padres 1095584-2202172030

7.3.3. Acceso secuencial al vector (recorrido)

Se puede acceder a los elementos de un vector para introducir datos *(escribir)* en él o bien para visualizar su contenido *(leer)*. A la operación de efectuar una acción general sobre todos los elementos de un vector se le denomina *recorrido* del vector. Estas operaciones se realizan utilizando estructuras repetitivas, cuyas variables de control (por ejemplo, I) se utilizan como subíndices del vector (por ejemplo, S[I]). El incremento del contador del bucle producirá el tratamiento sucesivo de los elementos del vector.

EJEMPLO 7.1

Lectura de veinte valores enteros de un vector denominado F.

Procedimiento 1

```
algoritmo leer_vector
tipo
   array[1..20] de entero : FINAL
var
   FINAL : F
inicio
   desde i ← 1 hasta 20 hacer
      leer(F[i])
   fin_desde
fin
```

La lectura de veinte valores sucesivos desde el teclado rellenará de valores el vector F, comenzando con el elemento F[1] y terminando en F[20]. Si se cambian los límites inferior y superior (por ejemplo, 5 y 10), el bucle de lectura sería:

```
desde i ← 5 hasta 10 hacer
   leer(F[i])
fin_desde
```

Procedimiento 2

Los elementos del vector se pueden leer también con bucles **mientras** o **repetir**.

```
i ← 1
mientras i <= 20 hacer
   leer(F[i])
   i ← i + 1
fin_mientras
```

o bien

```
i ← 1
repetir
   leer (F[i])
   i ← i + 1
hasta_que i > 20
```

La salida o escritura de vectores se representa de un modo similar. La estructura:

```
desde i ← 1 hasta i ← 20 hacer
   escribir(F[i])
fin_desde
```

visualiza todo el vector completo (un elemento en cada línea independiente).

EJEMPLO 7.2

Este ejemplo procesa un array PUNTOS, *realizando las siguientes operaciones; a) lectura del array, b) cálculo de la suma de los valores del array, c) cálculo de la media de los valores.*

Kristher Padres 1095564-2202172031

El *array* lo denominaremos PUNTOS, el límite superior del rango lo introduciremos por teclado y el límite inferior lo consideraremos 1.

```
algoritmo media_puntos
const
   LÍMITE = 40
tipo
   array[1..LÍMITE] de real : PUNTUACIÓN
var
   PUNTUACIÓN : PUNTOS
   real : suma, media
   entero : i
inicio
   suma ← 0
   escribir('Datos del array')
   desde i ← 1 hasta LÍMITE hacer
      leer(PUNTOS[i])
      suma ← suma + PUNTOS[i]
   fin_desde
   media ← suma / LÍMITE
   escribir('La media es', media)
fin
```

Se podría ampliar el ejemplo, en el sentido de visualizar los elementos del *array*, cuyo valor es superior a la media. A través de una estructura **desde** se podría realizar la operación, añadiéndole al algoritmo anterior.

```
escribir('Elementos del array superior a la media')
desde i ← 1 hasta LÍMITE hacer
   si PUNTOS[i] > media entonces
      escribir(PUNTOS[i])
   fin_si
fin_desde
```

Krasher Padres 1095564-2202172032

1095564220217203

EJEMPLO 7.3

Calcular la media de las estaturas de una clase. Deducir cuántos son más altos que la media y cuántos son más bajos que dicha media (figura 7.2).

Solución

Tabla de variables

n	número de estudiantes de la clase	: entera
H[1]...H[n]	estatura de los n alumnos	: real
i	contador de alumnos	: entera
MEDIA	media de estaturas	: real
ALTOS	alumnos de estatura mayor que la media	: entera
BAJOS	alumnos de estatura menor que la media	: entera
SUMA	totalizador de estaturas	: real

7.3.4. Actualización de un vector

La operación de actualizar un vector puede constar a su vez de tres operaciones elementales:

añadir elementos
insertar elementos
borrar elementos

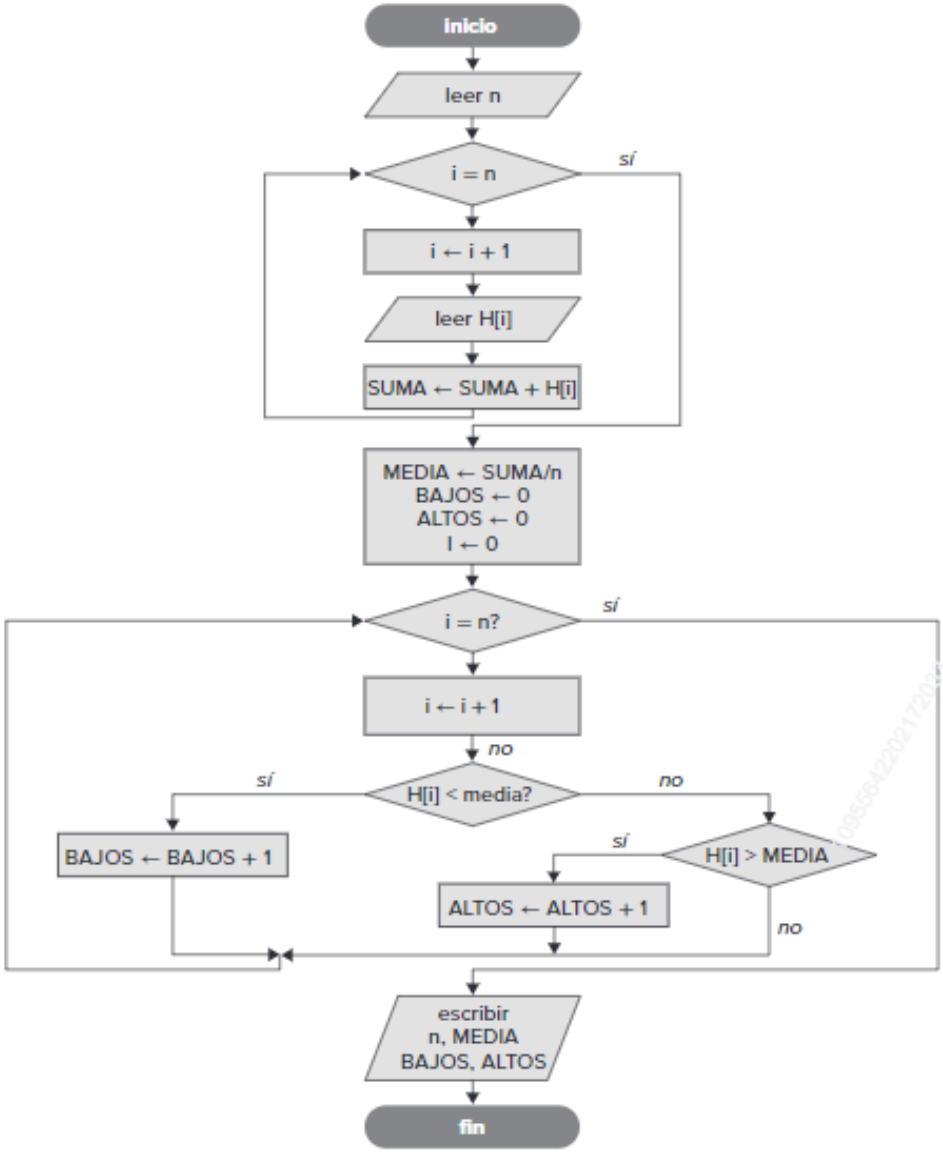


Figura 7.2. Diagrama de flujo para el cálculo de la estatura media de una clase.

Se denomina *añadir datos* a un vector la operación de añadir un nuevo elemento al final del vector. La única condición necesaria para esta operación consistirá en la comprobación de espacio de memoria suficiente para el nuevo vector; dicho de otro modo, que el vector no contenga todos los elementos con que fue definido al principio del programa.

EJEMPLO 7.4

Un array `TOTAL` *se ha dimensionado a seis elementos, pero sólo se le han asignado cuatro valores a los elementos* `TOTAL[1]`, `TOTAL[2]`, `TOTAL[3]` *y* `TOTAL[4]`. *Se podrán añadir dos elementos más con una simple acción de asignación.*

```
TOTAL[5] ← 14
TOTAL[6] ← 12
```

La *operación de insertar un elemento* consiste en introducir dicho elemento en el interior del vector. En este caso se necesita un desplazamiento previo hacia abajo para colocar el elemento nuevo en su posición relativa.

Krasner Padres 1095564-2202172033

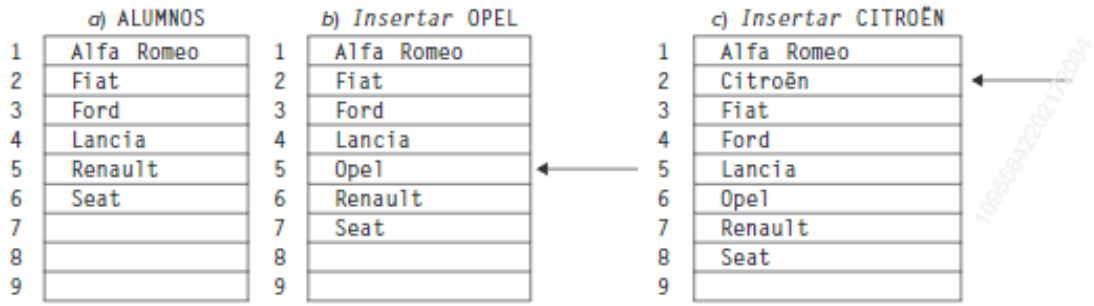
EJEMPLO 7.5

Se tiene un array Coches[2] *de nueve elementos que contiene siete marcas de automóviles en orden alfabético y se desea insertar dos nuevas marcas:* Opel *y* Citroën.

Como Opel está comprendido entre Lancia y Renault, se deberán desplazar hacia abajo los elementos 5 y 6, que pasarán a ocupar la posición relativa 6 y 7. Posteriormente debe realizarse la operación con Citroën, que ocupará la posición 2.

El algoritmo que realiza esta operación para un vector de *n* elementos es el siguiente, suponiendo que haya espacio suficiente en el vector.

```
1. //Calcular la posición ocupada por el elemento a insertar (por ejemplo, P)
2. //Inicializar contador de inserciones i ← n
3. mientras i >= P hacer
       //transferir el elemento actual i-ésimo hacia abajo, a la posición i+1
       COCHES[i + 1] ← COCHES[i]
       //decrementar contador
       i ← i - 1
       fin_mientras
 4. //insertar el elemento en la posición P COCHES[P] ←'nuevo elemento'
 5. //actualizar el contador de elementos del vector
 6. n ← n + 1
 7. fin
```



Si se deseara realizar más inserciones, habría que incluir una estructura de decisión **si-entonces** para preguntar si se van a realizar más inserciones.

La operación de borrar un elemento al final del vector no presenta ningún problema; el borrado de un elemento del interior del vector provoca el movimiento hacia arriba de los elementos inferiores a él para reorganizar el vector.

El algoritmo de borrado del elemento *j-ésimo* del vector COCHES es el siguiente:

```
algoritmo borrado
inicio
//se utilizará una variable auxiliar —AUX— que contendrá el valor
//del elemento que se desea borrar
   AUX ← COCHES[j]
   desde i ← j hasta N-1 hacer
      //llevar elemento j + 1 hacia arriba
      COCHES[i] ← COCHES[i + 1]
   fin_desde
   //actualizar contador de elementos
   //ahora tendrá un elemento menos, N - 1
   N ← N - 1
fin
```

[2] En Latinoamérica, su término equivalente es CARRO o AUTO.

Krasner Padres 1095564-2202172034

7.4. Arrays (arreglos) de varias dimensiones

Los vectores examinados hasta ahora se denominan *arrays* unidimensionales y en ellos cada elemento se define o referencia por un índice o subíndice. Estos vectores son elementos de datos escritos en una secuencia. Sin embargo, existen grupos de datos que son representados mejor en forma de tabla o matriz con dos o más subíndices. Ejemplos típicos de tablas o matrices son: tablas de distancias kilométricas entre ciudades, cuadros horarios de trenes o aviones, informes de ventas periódicas (mes/unidades vendidas o bien mes/ventas totales), etcétera. Se pueden definir *tablas* o *matrices* como *arrays multidimensionales*, cuyos elementos pueden referenciarse por dos, tres o más subíndices. Los *arrays* no unidimensionales los dividiremos en dos grandes grupos:

Arrays bidimensionales	(2 dimensiones)
Arrays multidimensionales	(3 o más dimensiones)

7.4.1. Arrays bidimensionales (tablas/matrices)

El *array bidimensional* puede considerarse como un vector de vectores. Es, por consiguiente, un conjunto de elementos, todos del mismo tipo, en el cual el orden de los componentes es significativo y en el que se necesita especificar dos subíndices para poder identificar cada elemento del *array*.

Si se visualiza un *array* unidimensional, puede considerarse como una columna de datos; un *array* bidimensional es un grupo de columnas, como se ilustra en la figura 7.3.

El diagrama representa una tabla o matriz de treinta elementos (5 × 6) con 5 filas y 6 columnas. Como en un vector de treinta elementos, cada uno de ellos tiene el mismo nombre. Sin embargo, un subíndice no es suficiente para especificar un elemento de un *array* bidimensional; por ejemplo, si el nombre del *array* es `M`, no puede indicarse `M[3]`, ya que no sabemos si es el tercer elemento de la primera fila o de la primera columna. Para evitar la ambigüedad, los elementos de un *array* bidimensional se referencian con dos subíndices: el primer subíndice se refiere a la *fila* y el segundo subíndice se refiere a la *columna*. Por consiguiente, `M[2, 3]` se refiere al elemento de la segunda fila, tercera columna. En nuestra tabla ejemplo `M[2, 3]` contiene el valor 18.

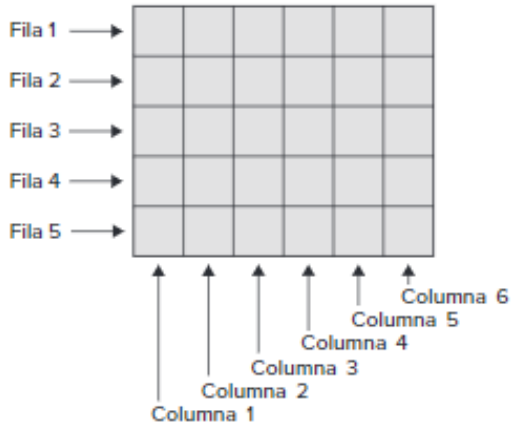


Figura 7.3. Array bidimensional.

Un *array bidimensional* `M`, también denominado *matriz* (términos matemáticos) o *tabla* (términos financieros), se considera que tiene dos dimensiones (una dimensión por cada subíndice) y necesita un valor para cada subíndice para poder identificar un elemento individual. En notación estándar, normalmente el primer subíndice se refiere a la fila del *array*, mientras que el segundo se refiere a la columna del *array*. Es decir, `B[I, J]` es el elemento de `B` que ocupa la I^a fila y la J^a columna, como se indica en la figura 7.4.

El elemento `B[I, J]` también puede representarse por `BI, J`. Más formalmente en notación algorítmica, el *array* `B` con elementos del tipo `T` (numéricos, alfanuméricos, etcétera) con *subíndices fila* que varían en el rango de `1` a `M` y *subíndices columna* en el rango de `1` a `N` es:

Krasner Podres 1095584-2202172125

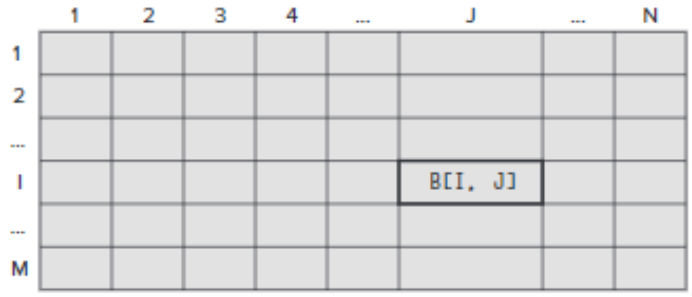


Figura 7.4. Elemento `B[I, J]` del array `B`.

```
B(1:M, 1:N) = (B[I, J])
```

donde `I = 1, ..., M` *o bien* `1 <= I <= M`
`J = 1, ..., N` `1 <= J <= N`

cada elemento `B[I, J]` *es de tipo T.*

El *array* `B` se dice que tiene `M` por `N` elementos. Existen `N` elementos en cada fila y `M` elementos en cada columna (M*N).

Los *arrays* de dos dimensiones son muy frecuentes: las calificaciones de los estudiantes de una clase se almacenan en una tabla `NOTAS` de dimensiones `NOTAS[20, 5]`, donde 20 es el número de alumnos y 5 el número de asignaturas. El valor del subíndice `I` debe estar entre 1 y 20, y el de `J` entre 1 y 5. Los subíndices pueden ser variables o expresiones numéricas, `NOTAS(M, 4)` y en ellos el subíndice de filas irá de 1 a `M` y el de columnas de 1 a `N`.

En general, se considera que un *array* bidimensional comienza sus subíndices en 0 o en 1 (según el lenguaje de programación, 0 en el lenguaje C, 1 en Fortran), pero pueden tener límites seleccionados por el usuario durante la codificación del algoritmo. En general, el *array* bidimensional `B` con su primer subíndice, variando desde un límite inferior `L` (inferior, *low*) a un límite superior `U` (superior, *up*). En notación algorítmica:

```
B(L1:U1, L2:U2) = (B[I, J])
```

donde `L1 <= I <= U1`
`L2 <= J <= U2`

cada elemento `B[I, J]` *es de tipo T.*

El número de elementos de una fila de `B` es `U2-L2+1` y el número de elementos en una columna de `B` es `U1-L1+1`.

Por consiguiente, el número total de elementos del *array* `B` es `(U2-L2+1)*(U1-L1+1)`.

EJEMPLO 7.6

La matriz T representa una tabla de notaciones de saltos de altura (primer salto), donde las filas representan el nombre del atleta y las columnas las diferentes alturas saltadas por el atleta. Los símbolos almacenados en la tabla son: x, salto válido; 0, salto nulo o no intentado.

Fila \ Columna T	2.00	2.10	2.20	2.30	2.35	2.40
García	x	0	x	x	x	0
Pérez	0	x	x	0	x	0
Gil	0	0	0	0	0	0
Mortimer	0	0	0	x	x	x

Krasner Podres 1095584-2202172125

EJEMPLO 7.7

Un ejemplo típico de un array bidimensional es un tablero de ajedrez. Se puede representar cada posición o casilla del tablero mediante un array, en el que cada elemento es una casilla y en el que su valor será un código representativo de cada figura del juego.

Figura 7.5. Array típico, "tablero de ajedrez".

Krasner Podres 1095584-2202172125

Los diferentes elementos serán:

`elemento[i, j] = 0`	si no hay nada en la casilla *[i, j]*
`elemento[i, j] = 1`	si el cuadro (casilla) contiene un peón blanco
`elemento[i, j] = 2`	un caballo blanco
`elemento[i, j] = 3`	un alfil blanco
`elemento[i, j] = 4`	una torre blanca
`elemento[i, j] = 5`	una reina blanca
`elemento[i, j] = 6`	un rey blanco

y los correspondientes números negativos para las piezas negras.

1095584220217125

EJEMPLO 7.8

Supongamos que se dispone de un mapa de ferrocarriles y los nombres de las estaciones (ciudades) están en un vector denominado "ciudad". El array `f` *puede tener los siguientes valores:*

`f[i, j] = 1`	si existe enlace entre las ciudades `i` y `j`, `ciudad[i]` y `ciudad[j]`
`f[i, j] = 0`	no existe enlace

Nota

El *array* f resume la información de la estructura de la red de enlaces.

7.5. Arrays (arreglos) multidimensionales

Un *array* puede ser definido de tres dimensiones, cuatro dimensiones, hasta de *n-dimensiones*. Los conceptos de rango de subíndices y número de elementos se pueden ampliar directamente desde *arrays* de una y dos dimensiones a estos *arrays* de orden más alto. En general, un *array* de n-dimensiones requiere que los valores de los *n* subíndices

puedan ser especificados a fin de identificar un elemento individual del *array*. Si cada componente de un *array* tiene *n* subíndices, el *array* se dice que es sólo de *n-dimensiones*. El *array* A de n-dimensiones se puede identificar como:

```
A(L1:U1, L2:U2, ... Ln:Un)
```

y un elemento individual del *array* se puede especificar por:

$A(I_1, I_2, \ldots, I_n)$

donde cada subíndice I_k está dentro de los límites adecuados:

$L_k <= I_k <= U_k$ donde $k = 1, 2, \ldots, n$

El número total de elementos de un *array* A es:

$$\prod_{k=1}^{n} (U_k - L_k + 1)$$ Π (símbolo del producto)

que puede escribirse alternativamente como:

$(U_1-L_1+1)*(U_2-L_2+1)*\ldots*(U_N-L_N+1)$

Si los límites inferiores comenzasen en 1, el *array* se representaría por:

$A(K_1, K_2, \ldots, K_n)$ *o bien* $A_{k1, k2, \ldots, kn}$

donde

$1 <= K_1 <= S_1$
$1 <= K_2 <= S_2$
.
.
$1 <= K_n <= S_n$

Krasner Podnes 1095564-2202172125

1095564220217212S

EJEMPLO 7.9

Un array de tres dimensiones puede ser uno que contenga los datos relativos al número de estudiantes de la universidad ALFA *de acuerdo con los siguientes criterios:*

- *Cursos (primero a quinto).*
- *Sexo (varón/hembra).*
- *Diez facultades.*

El *array* ALFA puede ser de dimensiones 5 por 2 por 10 (alternativamente 10 × 5 × 2 o 10 × 2 × 5, 2 × 5 × 10, etc.). La figura 7.6 representa el *array* ALFA.

El valor de elemento ALFA[I, J, K] es el número de estudiantes del curso I de sexo J de la facultad K. Para ser válido I debe ser 1, 2, 3, 4 o 5; J debe ser 1 o 2; K debe estar comprendida entre 1 y 10 inclusive.

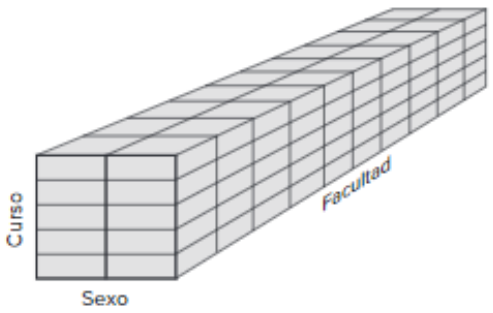


Figura 7.6. Array de tres dimensiones.

EJEMPLO 7.10

Otro array de tres dimensiones puede ser PASAJE que representa el estado actual del sistema de reserva de una línea aérea, donde:

```
i = 1, 2, ..., 10      representa el número de vuelo
j = 1, 2, ..., 60      representa la fila del avión
k = 1, 2, ..., 12      representa el asiento dentro de la fila
```

Entonces

```
pasaje[i, j, k] = 0        asiento libre
pasaje[i, j, k] = 1        asiento ocupado
```

7.6. Almacenamiento de arrays en memoria

Las representaciones gráficas de los diferentes *arrays* se recogen en la figura 7.7. Debido a la importancia de los *arrays*, casi todos los lenguajes de programación de alto nivel proporcionan medios eficaces para almacenar y acceder a los elementos de los *arrays*, de modo que el programador no tenga que preocuparse sobre los detalles específicos de almacenamiento. Sin embargo, el almacenamiento en la computadora está dispuesto fundamentalmente en secuencia contigua, de modo que cada acceso a una matriz o tabla la máquina debe realizar la tarea de convertir la posición dentro del *array* en una posición perteneciente a una línea.

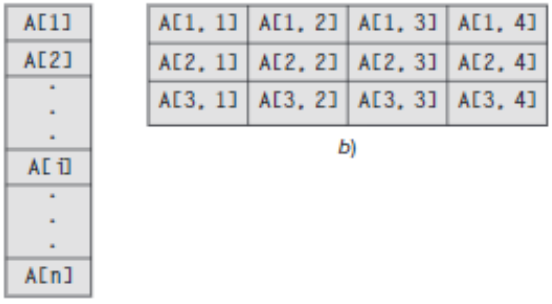


Figura 7.7. Arrays de una y dos dimensiones.

7.6.1. Almacenamiento de un vector

El almacenamiento de un vector en memoria se realiza en celdas o posiciones secuenciales. Así, en el caso de un vector A con un subíndice de rango 1 a *n*:

Posición B	A[1]
Posición B+1	A[2]
. . .	A[3]
	. . .
	A[i]
	. . .
Posición B+n–1	A[n]

Krasher Padres 1095584-2202172225

Si cada elemento del *array* ocupa `S` bytes (1 *byte* = 8 bits) y `B` es la dirección inicial de la memoria central de la computadora —*posición o dirección base*—, la dirección inicial del elemento *i-ésimo* sería:

```
B+(I-1)*S
```

Nota

Si el límite inferior no es igual a 1, considérese el *array* declarado como N(4:10); la dirección inicial de N(6) es B+(6–4)*S.

En general, el elemento `N(I)` de un *array* definido como `N(L:U)` tiene la dirección inicial

```
B+(I-L)*S
```

7.6.2. Almacenamiento de arrays multidimensionales

Debido a que la memoria de la computadora es lineal, un *array* multidimensional debe estar linealizado para su disposición en el almacenamiento. Los lenguajes de programación pueden almacenar los *arrays* en memoria de dos formas: *orden de fila mayor* y *orden de columna mayor*.

El medio más natural en que se leen y almacenan los *arrays* en la mayoría de los compiladores es el denominado *orden de fila mayor* (figura 7.8). Por ejemplo, si un *array* es `B[1:2, 1:3]`, el orden de los elementos en la memoria es:

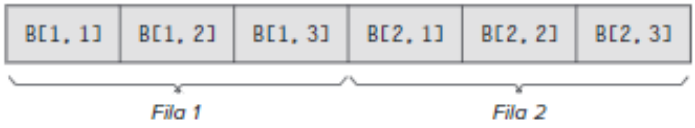


Figura 7.8. Orden de fila mayor.

C, Cobol y Pascal *almacenan los elementos por filas.*

Fortran emplea el *orden de columna mayor* en el que las entradas de la primera columna vienen primero.

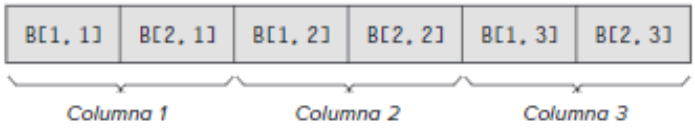


Figura 7.9. Orden de columna mayor.

De modo general, el compilador del lenguaje de alto nivel debe ser capaz de calcular con un índice *[i, j]* la posición del elemento correspondiente.

En un *array* en orden de fila mayor, cuyos subíndices máximos sean *m* y *n* (*m*, filas; *n*, columnas), la posición p del elemento [i, j] con relación al primer elemento es:

```
p = n(i-1)+j
```

Para calcular la dirección real del elemento [i, j] se añade p a la posición del primer elemento y se resta 1. La representación gráfica del almacenamiento de una tabla o matriz `B[2, 4]` y `C[2, 4]` (figura 7.10).

En el caso de un *array* de tres dimensiones, supongamos un *array tridimensional* `A[1:2, 1:4, 1:3]`. La figura 7.11 representa el *array* `A` y su almacenamiento en memoria.

Krasher Padres 1095564-2202172225

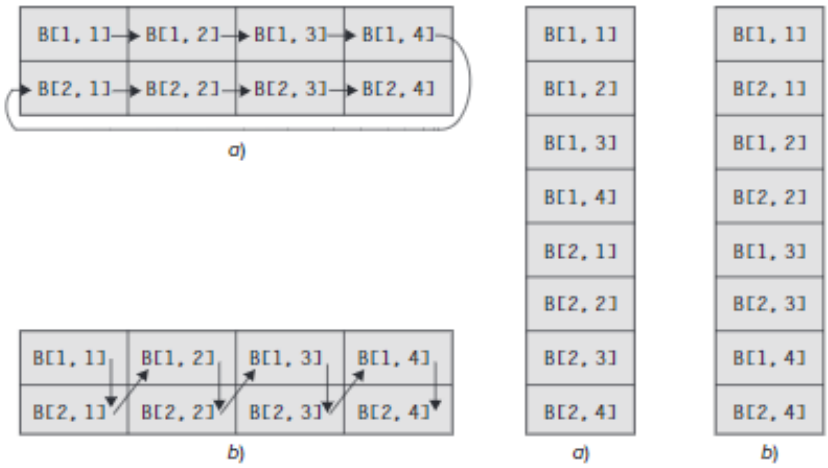


Figura 7.10. Almacenamiento de una matriz: *a*) por filas, *b*) por columnas.

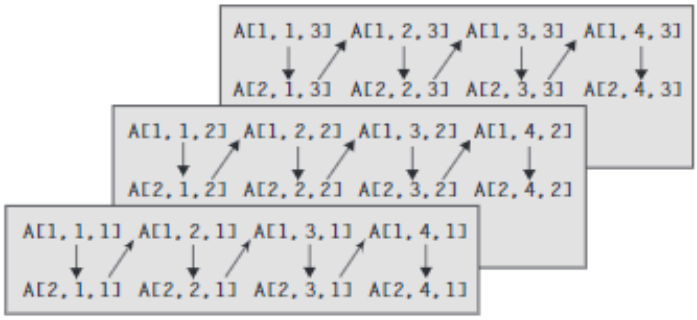


Figura 7.11. Almacenamiento de una matriz `A[2,4,3]` por columnas.

En orden a determinar si es más ventajoso almacenar un *array* en orden de columna mayor o en orden de fila mayor, es necesario conocer en qué orden se referencian los elementos del *array*. De hecho, los lenguajes de programación no le dan opción al programador para que elija una técnica de almacenamiento.

Consideremos un ejemplo del cálculo del valor medio de los elementos de un *array* `A` de 50 por 300 elementos, `A[50, 300]`. Los algoritmos de almacenamiento respectivos serán:

Almacenamiento por columna mayor

```
total ← 0
desde j ← 1 hasta 300 hacer
   desde i ← 1 hasta 50 hacer
      total ← total + a[i, j]
   fin_desde
fin_desde
media ← total / (300*50)
```

Almacenamiento por fila mayor

```
total ← 0
desde i ← 1 hasta 50 hacer
      desde j ← 1 hasta 300 hacer
      total ← total + a[i, j]
   fin_desde
fin_desde
media ← total / (300*50)
```

Krasher Padres 1095584-2202172225

7.7. Estructuras *versus* registros

Un *array* permite el acceso a una lista o una tabla de datos del mismo tipo utilizando un único nombre de variable. En ocasiones, sin embargo, se desea almacenar información de diferentes tipos, tales como un nombre de cadena, un número de código entero y un precio de tipo real (coma flotante) juntos en una única estructura. Una estructura que almacena diferentes tipos de datos bajo una misma variable se denomina *registro*.

En POO[3] el almacenamiento de información de diferentes tipos con un único nombre suele efectuarse en clases. No obstante, las clases son tipos referencia, esto significa que a los objetos de la clase se accede mediante una referencia. Sin embargo, en muchas ocasiones se requiere el uso de tipos valor. Las variables de un tipo valor contienen directamente los datos, mientras que las variables de tipos referencia almacenan una referencia al lugar donde se encuentran almacenados sus datos. El acceso a los objetos a través de referencia añade tareas y tiempos suplementarios y también consume espacio. En el caso de pequeños objetos este espacio extra puede ser significativo. Algunos lenguajes de programación como C y los orientados a objetos como C++, C#, ofrecen el tipo estructura para resolver estos inconvenientes. Una *estructura* es similar a una *clase* en orientación a objetos e igual a un registro en lenguajes estructurados como C pero es un tipo valor en lugar de un tipo referencia.

7.7.1. Registros

Un registro en **Pascal** es similar a una estructura en C y aunque en otros lenguajes como C# y C++ las clases pueden actuar como estructuras, en este capítulo restringiremos su definición al puro registro contenedor de diferentes tipos de datos. Un registro se declara con la palabra reservada **`estructura`** (***`struct`***, en inglés) o **`registro`** y se declara utilizando los mismos pasos necesarios para utilizar cualquier variable. Primero, se debe declarar el registro y a continuación se asignan valores a los miembros o elementos individuales del registro o estructura.

Krasher Padres 1995564-2202172225

Sintaxis

```
estructura: nombre_clase
    tipo_1: campo1
    tipo_2: campo2
    ...
fin_estructura
```

```
registro: nombre_tipo
    tipo_1: campo1
    tipo_2: campo2
    ...
fin_registro
```

Ejemplo

```
estructura: fechaNacimiento
   entero: mes // mes de nacimiento
   entero: día // día de nacimiento
   entero: año // año de nacimiento
fin_estructura
```

La declaración anterior reserva almacenamiento para los elementos de datos individuales denominados *campos* o **miembros** de la estructura. En el caso de fecha, la estructura consta de tres campos día, mes y año relativos a una fecha de nacimiento o a una fecha en sentido general. El acceso a los miembros de la estructura se realiza con el operador punto y con la siguiente sintaxis:

```
Nombre_estructura.miembro
```

Así `fechaNacimiento.mes` se refiere al miembro mes de la estructura fecha, y `fechaNacimiento.día` se refiere al día de nacimiento de una persona. Un tipo de dato estructura más general podría ser `Fecha` y que sirviera para

[3] Programación orientada a objetos.

cualquier dato aplicable a cualquier aplicación (fecha de nacimiento, fecha de un examen, fecha de comienzo de clases, etcétera).

```
estructura: Fecha
   entero: mes
   entero: día
   entero: año
fin_estructura
```

Declaración de tipos estructura

Una vez definido un tipo estructura se pueden declarar variables de ese tipo al igual que se hace con cualquier otro tipo de datos. Por ejemplo, la sentencia de definición:

```
Fecha: Cumpleaños, delDía
```

reserva almacenamiento para dos variables llamadas `Cumpleaños` y `delDía`, respectivamente. Cada una de estas estructuras individuales tiene el mismo formato que el declarado en la clase `Fecha`.

Los miembros de una estructura no están restringidos a tipos de datos enteros, sino que puede ser cualquier tipo de dato válido del lenguaje. Por ejemplo, consideremos un registro de un empleado de una empresa que constase de los siguientes miembros:

```
estructura Empleado
   Cadena: nombre
   entero: idNúmero
   real: Salario
   Fecha: FechaNacimiento
   entero: Antigüedad
fin_estructura
```

Observe que en la declaración de la estructura `Empleado`, el miembro `Fecha` es un nombre de un tipo estructura previamente definido. El acceso individual a los miembros individuales del tipo estructura de la clase `Empleado` se realiza mediante dos operadores punto, de la forma siguiente:

```
Empleado.Fecha.Día
```

y se refiere a la variable `Día` de la estructura `Fecha` de la estructura `Empleado`.

Estructuras de datos homogéneas y heterogéneas

Los registros (estructuras) y los *arrays* son tipos de datos estructurados. La diferencia entre estos dos tipos de estructuras de datos son los tipos de elementos que ellos contienen. Un *array* es una estructura de datos homogénea, que significa que cada uno de sus componentes deben ser del mismo tipo. Un registro es una estructura de datos heterogénea, que significa que cada uno de sus componentes pueden ser de tipos de datos diferentes. Por consiguiente, un *array* de registros es una estructura de datos cuyos elementos son de los mismos tipos heterogéneos.

7.8. Arrays (arreglos) de estructuras

La potencia real de una estructura o registro se manifiesta en toda su expresión cuando la misma estructura se utiliza para listas de datos. Por ejemplo, supongamos que se deben procesar los datos de la tabla de la figura 7.12.

Un sistema podría ser el siguiente: Almacenar los números de empleado en un *array* de enteros, los nombres en un *array* de cadenas de caracteres y los salarios en un *array* de números reales. Al organizar los datos de esta forma,

Krasner Podres 1095584-2202172226

Número de empleado	Nombre del empleado	Salario
97005	Mackoy, José Luis	1500
95758	Mortimer, Juan	1768
87124	Rodríguez, Manuel	2456
67005	Carrigan, Luis José	3125
20001	Mackena, Luis Miguel	2156
20020	García de la Cruz, Heraclio	1990
99002	Mackoy, María Victoria	2450
20012	González, Yiceth	4780
21001	González, Rina	3590
97005	Rodríguez, Concha	3574

Figura 7.12. Lista de datos.

cada columna de la figura 7.13 se considera como una lista independiente que se almacena en su propio *array*. La correspondencia entre elementos de cada empleado individual se mantiene almacenan los datos de un empleado en la misma posición de cada *array*.

La separación de cada lista completa en tres *arrays* individuales no es muy eficiente, ya que todos los datos relativos a un empleado se organizan juntos en un registro como se muestra en la figura 7.13. Utilizando una estructura, se mantiene la integridad de los datos de la organización y bastará un programa que maneje los registros para poder ser manipulados con eficacia. La declaración de un *array* de estructuras es similar a la declaración de un *array* de cualquier otro tipo de variable. En consecuencia, en el caso del archivo de empleados de la empresa se puede declarar el *array* de empleado con el nombre `Empleado` y el registro o estructura lo denominamos `RegistroNómina`.

```
estructura: RegistroNómina
   entero: NúmEmpleado
   cadena[30]: Nombre
   real: Salario
fin_estructura
```

Se puede declarar un array de estructuras `RegistroNómina` que permite representar toda la tabla anterior:

```
array [1..10] de RegistroNómina : Empleado
```

La sentencia anterior construye un array de diez elementos `Empleado`, cada uno de los cuales es una estructura de datos de tipo `RegistroNómina` que representa a un empleado de la empresa Aguas de Sierra Mágina. Observe que la creación de un array de diez estructuras tiene el mismo formato que cualquier otro array. Por ejemplo, la creación de un array de diez enteros denominado `Empleado` requiere la declaración:

```
array [1..10] de entero : Empleado
```

En realidad la lista de datos de empleado se ha representado mediante una lista de registros como se mostraba en la figura 7.13.

Número de empleado	Nombre del empleado	Salario
97005	Mackoy, José Luis	1500
95758	Mortimer, Juan	1768
87124	Rodríguez, Manuel	2456
67005	Carrigan, Luis José	3125
20001	Mackena, Luis Miguel	2156
20020	García de la Cruz, Heraclio	1990
99002	Mackoy, María Victoria	2450
20012	González, Yiceth	4780
21001	González, Rina	3590
97005	Collado, Concha	3574

Figura 7.13. Lista de registros.

Krasher Padres 1095564-2202172228

7.9. Uniones

Una **unión** es un tipo de dato derivado (estructurado) que contiene sólo uno de sus miembros a la vez durante la ejecución del programa. Estos miembros comparten el mismo espacio de almacenamiento; es decir, una unión comparte el espacio en lugar de desperdiciarlo en variables que no se están utilizando. Los miembros de una unión pueden ser de cualquier tipo y contener dos o más tipos de datos. La sintaxis para declarar un tipo `unión` es idéntica a la utilizada para definir un tipo `estructura`, excepto que la palabra `unión` sustituye a `estructura`:

Sintaxis

```
unión nombre
   tipo_dato1     identificador1
   tipo_dato2     identificador2
      ...
fin_unión
```

El número de bytes utilizado para almacenar una unión debe ser suficiente para almacenar el miembro más grande. Sólo se puede hacer referencia a un miembro a la vez y, por consiguiente, a un tipo de dato a la vez. En tiempo de ejecución, el espacio asignado a la variable de tipo `unión` no incluye espacio de memoria más que para un miembro de la unión.

Krasher Padres 1095564-2202172229

EJEMPLO

```
unión TipoPeso
   entero Toneladas
   real Kilos
   real Gramos
fin_unión
TipoPeso peso        // Declaración de una variable tipo unión
```

1095564-2202172229

En tiempo de ejecución, el espacio de memoria asignado a la variable `peso` no incluye espacio para tres componentes distintos; en cambio, `peso` puede contener uno de los siguientes valores: `entero` o `real`.

El acceso a un miembro de la unión se realiza con el operador de acceso a miembros (punto, `.`):

```
peso.Toneladas = 325
```

Una unión es similar a una estructura con la diferencia de que sólo se puede almacenar en memoria de modo simultáneo un único miembro o campo, al contrario que la estructura que almacena espacio de memoria para todos sus miembros.

7.9.1. Unión *versus* estructura

Una estructura se utiliza para definir un tipo de dato con diferentes miembros. Cada miembro ocupa una posición independiente de memoria.

```
estructura rectángulo
inicio
   entero: anchura
   entero: altura
fin_estructura
```

La estructura **rectángulo** se puede representar en memoria en la figura 7.14.

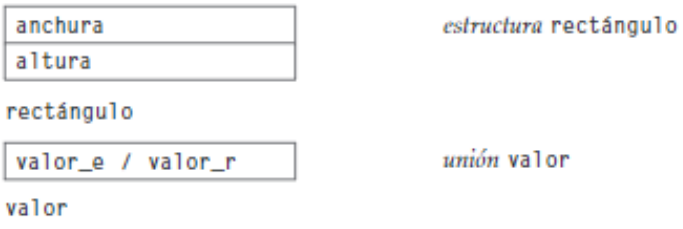


Figura 7.14. Estructura *versus* unión.

Una unión es similar a una estructura, sin embargo, sólo se define una única posición que puede ser ocupada por diferentes miembros con nombres diferentes:

```
unión valor
   entero valor_e
   real   valor_r
fin_unión
```

Los miembros `valor_e` y `valor_r` comparten el mismo espacio gráficamente; se puede pensar que una estructura es una caja con diferentes compartimentos, cada uno con su propio nombre (miembro), mientras que una unión es una caja sin compartimentos donde se pueden colocar diferentes etiquetas en su interior.

En una estructura, los miembros no interactúan; el cambio de un miembro no modifica a los restantes. En una unión todos los miembros ocupan el mismo espacio, de modo que sólo uno puede estar activo en un momento dado.

Krasner Padres 1095584-2202172230

EJERCICIO 7.1

Se desea almacenar información sobre una figura geométrica estándar (círculo, rectángulo o triángulo). La información necesaria para dibujar un círculo es diferente de los datos que se necesitan para dibujar un rectángulo, de modo que se necesitan diferentes estructuras para cada figura:

```
estructura círculo
   entero: radio
fin_estructura

estructura rectángulo
   entero: altura, anchura
fin_estructura

estructura triángulo
   entero: base
   entero: altura
fin_estructura
```

El ejercicio consiste en definir una estructura que pueda contener una figura genérica. El primer código es un número que indica el tipo de figura y el segundo es una unión que contiene la información de la figura.

```
estructura figura
entero: tipo      //tipo=0, círculo; tipo=1, rectángulo; tipo=2, triángulo
   union figura_genérica
      círculo: datos_círculo
      rectángulo: datos_rectángulo
      triángulo: datos_triángulo
   fin_union: datos
fin_estructura
```

De este modo se puede acceder a miembros de la unión, estructura específica o estructura general con el operador punto. Así el tipo de dato básico figura se puede definir y acceder a sus miembros de la forma siguiente:

```
figura: una_figura
// ...
una_figura.tipo ← 0
una_figura.datos.datos_circulo.radio ← 125
```

7.10. Enumeraciones

Una de las características importantes de la mayoría de los lenguajes de programación modernos es la posibilidad de definir nuevos tipos de datos. Entre estos tipos definidos por el usuario se encuentran los *tipos enumerados* o *enumeraciones*.

Un tipo enumerado o de enumeración es un tipo cuyos valores están definidos por una lista de constantes de tipo entero; las constantes se representan por identificadores separados por comas y encerrados entre llaves. Los valores de un tipo enumerado comienzan con 0, a menos que se especifique lo contrario y se incremente en 2. La sintaxis es:

```
enum nombre_tipo {identificador1, identificador2, ...}
```

identificador debe ser válido (**1ª**, **'B'**, **'24x'** no son identificadores válidos).

Krasner Podres 1095564-2202172231

EJEMPLO

```
enum Días {LUN, MAR, MIÉ, JUE, VIE, SÁB, DOM}
enum Meses {ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV DIC}
```

El tipo `Días` toma 7 valores, 0 a 6, y `Meses` toma 12 valores de 0 a 11. Estas declaraciones crean un nuevo tipo de datos, `Días` y `Meses`; los valores comienzan en 0, a menos que se indique lo contrario.

```
enum MESES {ENE←1, FEB, MAR, ABR, MAY, JUN, JUL, AGO←8, SEP, OCT, NOV, DIC}
```

Con la declaración anterior los meses se enumeran de 1 a 12.

El valor de cada constante de enumeración se puede establecer explícitamente en la definición, asignándole un valor al identificador, que puede ser el mismo o distinto entero. También se puede representar el tipo de dato con esta sintaxis:

```
enum Mes
{
   ENE ← 31, FEB ← 28, MAR ← 31, ABR ← 30, MAY ← 31, JUN ← 30, JUL ← 31, AGO ← 31,
   SEP ← 30, OCT ← 31, NOV ← 30, DIC ← 31
}
```

Si no se especifica ningún valor numérico, los identificadores en una definición de un tipo de enumeración se les asignan valores consecutivos que comienzan por cero.

EJEMPLO

```
enum Dirección { NORTE ← 0, SUR ← 1, ESTE ← 2, OESTE ← 3}
```

es equivalente a:

```
enum Dirección { NORTE, SUR, ESTE, OESTE}
```

Sintaxis

```
enum <nombre> {<enumerador1>, <enumerador2>,…}
enumerador identificador = expresión constante
```

Las variables de tipo enumeración se pueden utilizar en diferentes tipos de operaciones.

Creación de variables

```
enum Semáforo {verde, rojo, amarillo}
```

Se pueden asignar variables de tipo Semáforo:

```
var
   Semáforo Calle, Carretera, Plaza
```

Se crean las variables Calle, Carretera y Plaza de tipo Semáforo.

Asignación

La sentencia de asignación

```
Calle ← Rojo
```

no asigna a Calle la cadena de caracteres Rojo ni el contenido de una variable de nombre Rojo, sino que asigna el valor Rojo que es de uno de los valores del dominio del tipo de datos Semáforo.

Krasher Padres 1095584-2202172231

Sentencias de selección caso_de (switch), si-entonces (if-then)

Algoritmo

```
enum Mes { ENE, FEB, MAR, ... }
algoritmo DemoEnum
var Mes MesVacaciones
inicio
MesVacaciones ← ENE
si (MesVacaciones ← ENE)
   Escribir ('El mes de vacaciones es Enero')
fin_si
fin
```

Algoritmo

Se pueden usar valores de enumeración en una sentencia según_sea (switch):

```
tipo
enum Animales {Ratón, Gato, Perro, Paloma, Reptil, Canario}
var
   Animales: ADomésticos
   según_sea: ADomésticos
      Ratón: escribir '...'
      Gato  : escribir '...'
fin_según_sea
```

Sentencias repetitivas

Las variables de enumeración se pueden utilizar en bucles desde, mientras, ...:

```
algoritmo demoEnum2
tipo
enum meses { ENE=1, FEB, MAR, ABR, MAY, JUN, JUL. AGO, SEP, OCT, NOV, DIC}
```

```
var
   enum meses: mes
   inicio
   desde mes ← ENE hasta mes <= DIC
   ...
   fin_desde
fin
```

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

7.1. *Escribir un algoritmo que permita calcular el cuadrado de los cien primeros números enteros y a continuación escribir una tabla que contenga dichos cien números cuadrados.*

Solución

El problema consta de dos partes:

1. Cálculo de los cien primeros números enteros y sus cuadrados.
2. Diseño de una tabla T, T(1), T(2), ..., T(100) que contiene los siguientes valores:

```
T(1) = 1*1 = 1
T(2) = 2*2 = 4
T(3) = 3*3 = 9
...
```

El algoritmo se puede construir con estructuras de decisión o alternativas, o bien con estructuras repetitivas. En nuestro caso utilizaremos una estructura repetitiva desde.

```
algoritmo cuadrados
tipo
   array[1..100] de entero : tabla
var
   tabla : T
   entero : I, C
inicio
   desde I ← 1 hasta 100 hacer
      C ← I * I
      escribir(I, C)
   fin_desde
   desde I ← 1 hasta 100 hacer
      T[I] ← I * I
      escribir(T[I])
   fin_desde
fin
```

7.2. *Se tienen N temperaturas. Se desea calcular su media y determinar entre todas ellas cuáles son superiores o iguales a esa media.*

Solución

Análisis

En un primer momento se leen los datos y se almacenan en un vector (array unidimensional) TEMP(1:N).

A continuación se van realizando las sumas sucesivas a fin de obtener la media.

Por último, con un bucle de lectura de la tabla se va comparando cada elemento de la misma con la media y luego, mediante un contador, se calcula el número de temperaturas igual o superior a la media.

Krasher Podres 1095584-2202172233

Tabla de variables

N	Número de elementos del vector o tabla.
TEMP	Vector o tabla de temperatura.
SUMA	Sumas sucesivas de las temperaturas.
MEDIA	Media de la tabla.
C	Contador de temperaturas >= MEDIA.

Pseudocódigo

```
algoritmo temperaturas
const
   N = 100
tipo
   array[1..N] de real : temperatura
var
   temperatura: Temp
   entero : I, C
   real : suma, media
inicio
   suma ← 0
   media ← 0
   C ← 0
   desde I ← 1 hasta N hacer
      leer(Temp[I])
      suma ← suma+Temp[I]
   fin_desde
   media ← suma/N
   para I ← 1 hasta N hacer
      si Temp[I] >= media entonces
         C ← C+1
         escribir(Temp[I])
      fin_si
   fin_para
   escribir('La media es:', media)
   escribir('El total de temperaturas >=', media, 'es:', C)
fin
```

7.3. *Escribir el algoritmo que permita sumar el número de elementos positivos y el de negativos de una tabla T.*

Solución

Sea una tabla T de dimensiones M, N leídas desde el teclado.

Tabla de variables

I, J, M, N:	entero
SP:	real
SN:	real

Pseudocódigo

```
algoritmo suma_resta
const
   M = 50
   N = 20
```

Krasner Podres 1095584-2202172237

```
   tipo
      array[1..M, 1..N] de real : Tabla
   var
      Tabla : T
      entero : I, J
      real : SP, SN
   inicio
      SP ← 0
      SN ← 0
      desde I ←1 hasta M hacer
         desde J ←1 hasta N hacer
            si T[I, J] > 0 entonces
               SP ←SP + T[I, J]
            si_no
               SN ←SN + T[I, J]
            fin_si
         fin_desde
      fin_desde
   escribir('Suma de positivos', SP, 'de negativos', SN)
fin
```

7.4. *Inicializar una matriz de dos dimensiones con un valor constante dado K.*

Solución

Análisis

El algoritmo debe tratar de asignar la constante K a todos los elementos de la matriz `A[M, N]`.

```
A[1, 1] = K   A[1, 2] = K ... A[1, N] = K
.
.
A[M, 1] = K   A[M, 2] = K ... A[M, N] = K
```

Dado que es una matriz de dos dimensiones, se necesitan dos bucles anidados para la lectura.

Pseudocódigo

```
algoritmo inicializa_matriz
inicio
   desde I ←1 hasta M hacer
      desde J ←1 hasta N hacer
         A[I, J] ←K
      fin_desde
   fin_desde
fin
```

7.5. *Realizar la suma de dos matrices bidimensionales.*

Solución

Análisis

Las matrices `A[I, J]`, `B[I, J]` para que se puedan sumar deben tener las mismas dimensiones. La matriz suma `S[I, J]` tendrá iguales dimensiones y cada elemento será la suma de las correspondientes matrices `A` y `B`:

```
S[I, J] = A[I, J] + B[I, J]
```

Dado que se trata de matrices de dos dimensiones, el proceso se realizará con dos bucles anidados.

Krasher Padres 1095564-2202172238

Pseudocódigo

```
algoritmo suma_matrices
inicio
   desde I ←1 hasta N hacer
      desde J ←1 hasta M hacer
         S[I, J] ←A[I, J] + B[I, J]
      fin_desde
   fin_desde
fin
```

7.6. *Se dispone de una tabla T de dos dimensiones. Calcular la suma de sus elementos.*

Solución

Supongamos las dimensiones de `T`, `M` y `A` y que se compone de números reales.

Tabla de variables

`I`	Contador de filas.
`J`	Contador de columnas.
`M`	Número de filas de la tabla `T`.
`N`	Número de columnas de la tabla `T`.
`T`	Tabla.
`S`	Suma de los elementos de la tabla.
`I, J, M, N`	Enteros.
`T, S`	Reales.

Pseudocódigo

```
algoritmo suma_elementos
const
   M = 50
   N = 20
tipo
   array[1..M, 1..N] de real : Tabla
var
   entero : I, J
   Tabla : T
   real : S
inicio
   desde I ← 1 hasta M hacer
      desde J ← 1 hasta N hacer
         leer(T[I, J])
      fin_desde
   fin_desde
   S ←  0       {inicialización de la suma S}
   desde I ← 1 hasta M hacer
      desde J ← 1 hasta N hacer
         S ← S + T[I, J]
      fin_desde
   fin_desde
   escribir('La suma de los elementos de la matriz =', S)
fin
```

Krasner Padres 1095564-2202172239

7.7. *Realizar la búsqueda de un determinado nombre en una lista de nombres, de modo que el algoritmo imprima los siguientes mensajes según el resultado:*

'Nombre encontrado'	*si el nombre está en la lista*
'Nombre no existe'	*si el nombre no está en la lista*

Solución

Se recurrirá en este ejercicio a utilizar un interruptor SW, de modo que si SW = falso el nombre no existe en la lista y si SW = verdadero el nombre existe en la lista (o bien, en caso de no existir la posibilidad de variables lógicas, definir SW como SW = 0 si es falso y SW = 1 si es verdadero o cierto).

Método 1

```
algoritmo búsqueda
const
   N = 50
tipo
   array[1..N] de cadena : Listas
var
   Listas : l
   lógico : SW
   cadena : nombre
   entero : I
inicio
   SW ← falso
   leer(nombre)
   desde I ← 1 hasta N hacer
      si l[I] = nombre entonces
         SW ← verdadero
      fin_si
   fin_desde
   si SW entonces
      escribir('Encontrado')
   si_no
      escribir('No existe', nombre)
   fin_si
fin
```

Método 2

```
algoritmo búsqueda
const
   N = 50
tipo
   array[1..N] de cadena : Listas
var
   Listas : l
   lógico : SW
   cadena : nombre
   entero : I
inicio
   SW ← 0
   leer(nombre)
   desde I ← 1 hasta N hacer
      si l[I] = nombre entonces
         SW ← 1
```

Krasner Padres 1095564-2202172240

```
      fin_si
   fin_desde
   si SW = 1 entonces
      escribir('Encontrado')
   si_no
      escribir('No existe', nombre)
       fin_si
   fin
```

7.8. *Se desea permutar las filas I y J de una matriz (array) de dos dimensiones (M*N):M filas, N columnas.*

Solución

Análisis

La tabla `T(M*N)` se puede representar por:

```
T[1, 1]   T[1, 2]   T[1, 3]    ...     T[1, N]
T[2, 1]   T[2, 2]   T[2, 3]    ...     T[2, N]
.
.
T[M, 1]   T[M, 2]   T[M, 3]    ...     T[M, N]
```

El sistema para permutar globalmente toda la fila I con la fila J se debe realizar permutando uno a uno el contenido de los elementos `T[I, K]` y `T[J, K]`.

Para intercambiar entre sí los valores de dos variables, recordemos que se necesitaba una variable auxiliar `AUX`. Así, para el caso de las variables `A` y `B`:

```
AUX ← A
 A  ← B
 B  ← AUX
```

En el caso de nuestro ejercicio, para intercambiar los valores `T[I, K]` y `T[J, K]` se debe utilizar el algoritmo:

```
AUX      ← T[I, K]
T[I, K]  ← T[J, K]
T[J, K]  ← T[I, K]
```

Tabla de variables

I, J, K, M, N	Enteras
AUX	Real
Array	Real

Pseudocódigo

```
algoritmo intercambio
const
   M = 50
   N = 30
tipo
   array[1..M, 1.. N] de entero : Tabla
var
   Tabla : T
   entero : AUX, I, J, K
inicio
```

Krasher Padres 1095564-2202172241

```
   {En este ejercicio y dado que ya se han realizado muchos ejemplos de
   lectura de arrays con dos bucles desde, la operación de lectura completa del array se
   representará con la instrucción de leerArr(T)}
   leerArr(T)
   //Deducir I, J a intercambiar
   leer(I, J)
   desde K ← I hasta N hacer
      AUX ← T[I, K]
      T[I, K] ← T[J, K]
      T[J, K] ← AUX
   fin_desde
   //Escritura del nuevo array
   escribirArr(T)
fin
```

7.9. *Algoritmo que nos permita calcular la desviación estándar (SIGMA) de una lista de N números (N <= 15)*

Sabiendo que

$$\text{DESVIACIÓN} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - m)^2}{n-1}}$$

```
algoritmo Calcular_desviación
   tipo
      array[1..15] de real : arr
var
      arr : x
      entero : n
inicio
   llamar_a leer_array(x, n)
   escribir('La desviación estándar es',desviación(x, n))
fin

procedimiento leer_array(S arr:x       S entero:n)
   var
      entero : i
   inicio
      repetir
         escribir('Diga número de elementos de la lista')
         leer(n)
      hasta_que n <= 15
      escribir('Deme los elementos:')
      desde i ← 1 hasta n hacer
         leer(x[i])
      fin_desde
   fin_procedimiento

real función desviación(E arr : x     E entero : n)
   var
      real   : suma, xm, sigma
      entero : i
   inicio
      suma ← 0
      desde i ← 1 hasta n hacer
      suma ← suma + x[i]
```

Krasher Podres 1095564-2202172242

```
   fin_desde
   xm ← suma / n
   sigma ← 0
   desde i ← 1 hasta n hacer
      sigma ← sigma + cuadrado (x[i] - xm)
   fin_desde
   devolver(raiz2 (sigma / (n-1)))
fin_función
```

7.10. *Utilizando arrays, escribir un algoritmo que visualice un cuadrado mágico de orden impar n, comprendido entre 3 y 11. El usuario debe elegir el valor de n.*

Un cuadrado mágico se compone de números enteros comprendidos entre 1 y n. La suma de los números que figuran en cada fila, columna y diagonal son iguales.

Ejemplo

8	1	6
3	5	7
4	9	2

Un método de generación consiste en situar el número 1 en el centro de la primera fila, el número siguiente en la casilla situada por encima y a la derecha y así sucesivamente. El cuadrado es cíclico, la línea encima de la primera es de hecho la última y la columna a la derecha de la última es la primera. En el caso de que el número generado caiga en una casilla ocupada, se elige la casilla que se encuentre debajo del número que acaba de ser situado.

```
algoritmo Cuadrado_mágico
   var    entero : n
inicio
   repetir
      escribir('Dame las dimensiones del cuadrado (3 a 11)')
      leer(n)
   hasta_que (n mod 2 <>0) Y (n <= 11) Y (n >= 3)
   dibujarcuadrado(n)
fin
procedimiento dibujarcuadrado(E entero:n)
   var    array[1..11,1..11] de entero : a
          entero                       : i,j,c
   inicio
      ← 2
      ← n div 2
      desde c ← 1 hasta n*n hacer
         i ← i - 1
         j ← j + 1
         si j > n entonces
            j ← 1
         fin_si
         si i < 1 entonces
            i ← n
         fin_si
         a[i,j] ← c
         si c mod n = 0 entonces
            j ← j - 1
            i ← i + 2
```

Krasher Padres 1095564-2202172243

```
      fin_si
   fin_desde
   desde i ← 1 hasta n hacer
      desde j ← 1 hasta n hacer
         escribir(a[i,j])
         [al codificar esta instrucción en un lenguaje será conveniente utilizar el
         parámetro correspondiente de "no avance de línea" en la salida en pantalla
         o impresora]
   fin_desde
      escribir(NL)
      //NL representa Nueva Línea, es decir, avance de línea
   fin_desde
fin_procedimiento
```

7.11. *Obtener un algoritmo que efectúe la multiplicación de dos matrices A, B.*

$A \in M_{m,p}$ *elementos*
$B \in M_{p,n}$ *elementos*

Matriz producto: $C \in M_{m,n}$ elementos, tal que

$$C_{i,j} = \sum_{i=1}^{p} a_{i,k} * b_{k,j}$$

```
algoritmo Multiplicar_matrices
   tipo array[1..10,1..10] de real : arr
   var  entero : m, n, p
        arr    : a ,b, c
   inicio
   repetir
      escribir('Dimensiones de la 1ª matriz (filas columnas)')
      leer(m,p)
      escribir('Columnas de la 2ª matriz')
      leer(n)
   hasta_que (n <= 10) Y (m <= 10) Y (p <= 10)
   escribir ('Deme elementos de la 1ª matriz')
   llamar_a leer_matriz(a,m,p)
   escribir ('Deme elementos de la 2ª matriz')
   llamar_a leer_matriz(b,p,n)
   llamar_a calcescrproducto(a, b, c, m, p, n)
fin

procedimiento leer_matriz(S arr:matriz;E entero:filas,columnas)
var entero : i, j
inicio
   desde i ← 1 hasta filas hacer
      escribir('Fila ',i,':')
      desde j ← 1 hasta columnas hacer
         leer(matriz[i,j])
      fin_desde
```

Krasner Padres 1995564-2202172244

```
   fin_desde
fin_procedimiento

procedimiento calcescrproducto(E arr: a, b, c; E entero: m,p,n)
   var entero : i, j, k

inicio
   desde i ←1 hasta m hacer
      desde j ←1 hasta n hacer
            c[i,j] ←0
         desde k ←1 hasta p hacer
            c[i,j] ←c[i,j] + a[i,k] * b[k,j]
         fin_desde
         escribir (c[i,j])     //no avanzar línea
      fin_desde
      escribir ( NL )          //avanzar línea, nueva línea
   Fin_desde
Fin_procedimiento
```

7.12. *Algoritmo que triangule una matriz cuadrada y halle su determinante. En las matrices cuadradas el valor del determinante coincide con el producto de los elementos de la diagonal de la matriz triangulada, multiplicado por –1 tantas veces como hayamos intercambiado filas al triangular la matriz.*

Proceso de triangulación de una matriz para todo i desde 1 hasta n – 1 hacer:

a) Si el elemento de lugar (i,i) es nulo, intercambiar filas hasta que dicho elemento sea no nulo o agotar los posibles intercambios.

b) A continuación se busca el primer elemento no nulo de la fila i-ésima y, en el caso de existir, se usa para hacer ceros en la columna de abajo.
Sea dicho elemento matriz[i,r]
Multiplicar fila i por matriz[i+1,r]/matriz[i,r] y restarlo a la i+1
Multiplicar fila i por matriz[i+2,r]/matriz[i,r] y restarlo a la i+2
.....
Multiplicar fila i por matriz[m,r]/matriz[i,r] y restarlo a la m

```
algoritmo Triangulación_matriz
Const m = <expresión>
      n = <expresión>
tipo array[1..m, 1..n] de real : arr
var arr : matriz
   real : dt
inicio
   llamar_a leer_matriz(matriz)
   llamar_a triangula(matriz, dt)
   escribir('Determinante=', dt)
fin

procedimiento leer_matriz (S arr : matriz)
   var entero: i, j

inicio
   escribir('Deme los valores para la matriz')

   desde i ←1 hasta m hacer
      desde j ←1 hasta n hacer
```

Krasner Podnes 1095584-2202172300

```
            leer(matriz[i, j])
         fin_desde
      fin_desde
   fin_procedimiento

procedimiento escribir_matriz (E arr : matriz)
   var entero    : i, j
      carácter : c
   inicio
      escribir('Matriz triangulada')
      desde i ← 1 hasta m hacer
         desde j ← 1 hasta n hacer
            escribir(matriz[i, j])    //no avanzar línea
         fin_desde
         escribir(NL)                 //avanzar línea, nueva línea
      fin_desde
      escribir('Pulse tecla para continuar')
      leer(c)
fin_procedimiento

procedimiento interc(E/S real: a,b)
   var real : auxi
   inicio
      auxi ← a
      a ← b
      b ← auxi
   fin_procedimiento

procedimiento triangula (E arr : matriz; S real dt)
   var entero: signo
       entero: t, r, i, j
       real: cs
   inicio
      signo ← 1
      desde i ← 1 hasta m - 1 hacer
         t ← 1
         si matriz[i, i] = 0 entonces
            repetir
               si matriz[i + t, i] <> 0 entonces
                  signo ← signo * (-1)
                  desde j ← 1 hasta n hacer
                    llamar_a interc(matriz[i,j],matriz[i + t,j])
                  fin_desde
                  llamar_a escribir_matriz(matriz)
               fin_si
               t ← t + 1
            hasta_que (matriz[i, i] <> 0) O (t = m - i + 1)
         fin_si
         r ← i - 1
         repetir
            r ← r + 1
```

Krasher Padres 1995564-2202172301
1995564-2202172301

```
      hasta_que (matriz[i, r] <> 0) O (r = n)
      si matriz[i, r] <> 0 entonces
         desde t ← i + 1 hasta m hacer
            si matriz[t, r] <> 0 entonces
               cs ← matriz[t, r]
               desde j ← r hasta n hacer
                  matriz[t, j] ← matriz[t, j] - matriz[i, j] * (cs / matriz[i, r])
               fin_desde
               llamar_a escribir_matriz(matriz)
            fin_si
         fin_desde
      fin_si
   fin_desde
   dt ← signo
   desde i ← 1 hasta m hacer
      dt ← dt * matriz[i, i]
   fin_desde
fin_procedimiento
```

Krasner Podres 1095584-2202172303

CONCEPTOS CLAVE

- *Array* bidimensional
- *Array* de una dimensión
- *Array* multidimensional
- *Arrays* como parámetros
- Arreglo (*array*)
- Datos estructurados
- Estructura
- Índice
- Lista
- Longitud de un *array*
- Subíndice
- Tabla
- Tamaño de un *array*
- Variable indexada
- Vector

RESUMEN

Un *array* (vector, lista o tabla) es una estructura de datos que almacena un conjunto de valores, todos del mismo tipo de datos. Un *array* de una dimensión, también conocido como *array* unidimensional o vector, es una lista de elementos del mismo tipo de datos que se almacenan utilizando un único nombre. En esencia, un *array* (arreglo) es una colección de variables que se almacenan en orden en posiciones consecutivas en la memoria de la computadora. Dependiendo del lenguaje de programación el índice del *array* comienza en 0 (lenguaje **C**) o bien en 1 (lenguaje **Fortran**); este elemento se almacena en la posición con la dirección más baja.

1. Un array unidimensional (vector o lista) es una estructura de datos que se puede utilizar para almacenar una lista de valores del mismo tipo de datos. Tales arrays se pueden declarar dando el tipo de datos de los valores que se van a almacenar y el tamaño del *array*. Por ejemplo, en **C/C++** la declaración:

   ```
   int núm[100]
   ```

   crea un array de 100 elementos, el primer elemento es `núm[0]` y el último elemento es `núm[99]`.
2. Los elementos del *array* se almacenan en posiciones contiguas en memoria y se referencian utilizando el nombre del *array* y un subíndice; por ejemplo, **`núm[25]`**. Cualquier expresión de valor entero no negativo se puede utilizar como subíndice y el subíndice 0 (en el caso de **C**) o 1 (caso de **Fortran**) siempre se refieren al primer elemento del *array*.

3. Se utilizan *arrays* para almacenar grandes colecciones de datos del mismo tipo. Esencialmente en los casos siguientes:
   - Cuando los elementos individuales de datos se deben utilizar en un orden aleatorio, como es el caso de los elementos de una lista o los datos de una tabla.
   - Cuando cada elemento representa una parte de un dato compuesto, tal como un vector, que se utilizará repetidamente en los cálculos.
   - Cuando los datos se deben procesar en fases independientes, como puede ser el cálculo de una media aritmética o varianza.
4. Un *array* de dos dimensiones (tabla) se declara listando el tamaño de las filas y de las columnas junto con el nombre del *array* y el tipo de datos que contiene. Por ejemplo, las declaraciones en **C** de:

```
int tabla1[5][10]
```

crean un *array* bidimensional de cinco filas y 10 columnas de tipo entero.

5. *Arrays paralelos*. Una tabla multicolumna se puede representar como un conjunto de *arrays* paralelos, un *array* por columna, de modo que todos tengan la misma longitud y se accede utilizando la misma variable de subíndice.
6. Los *arrays* pueden ser, de modo completo o por elementos, pasados como parámetros a funciones y a su vez ser argumentos de funciones.
7. La longitud de un *array* se fija en su declaración y no puede ser modificada sin una nueva declaración. Esta característica los hace a veces poco adecuados para aplicaciones que requieren de tamaños o longitudes variables.

Krasner Podres 1095584-2202172304

EJERCICIOS

7.1. Determinar los valores de I, J, después de la ejecución de las instrucciones siguientes:

```
var
   entero : I, J
   array[1..10] de entero : A
inicio
   I ← 1
   J ← 2
   A[I] ← J
   A[J] ← I
   A[J+I] ← I + J
   I ← A[I] + A[J]
   A[3] ← 5
   J ← A[I] - A[J]
fin
```

7.2. Escribir el algoritmo que permita obtener el número de elementos positivos de una tabla.

7.3. Rellenar una matriz identidad de 4 × 4.

7.4. Leer una matriz de 3 × 3 elementos y calcular la suma de cada una de sus filas y columnas, dejando dichos resultados en dos vectores, uno de la suma de las filas y otro de las columnas.

7.5. Cálculo de la suma de todos los elementos de un vector, así como la media aritmética.

7.6. Calcular el número de elementos negativos, cero y positivos de un vector dado de sesenta elementos.

7.7. Calcular la suma de los elementos de la diagonal principal de una matriz (4 × 4).

7.8. Se dispone de una tabla T de cincuenta números reales distintos de cero. Crear una nueva tabla en la que todos sus elementos resulten de dividir los elementos de la tabla T por el elemento T[K], siendo K un valor dado.

7.9. Se dispone de una lista (vector) de N elementos. Se desea diseñar un algoritmo que permita insertar el valor x en el lugar k-ésimo de la mencionada lista.

7.10. Se desea realizar un algoritmo que permita controlar las reservas de asientos de un vuelo MADRID-CARACAS, de acuerdo con las siguientes normas de la compañía aérea:

Número de asientos del avión: 300.
Asientos numeradas de 1 a 100: fumadores.
Asientos numeradas de 101 a 300: no fumadores.

Se debe realizar la reserva a petición del pasajero y cerrar la reserva cuando no haya plazas libres o el avión esté próximo a despegar. Como

ampliación de este algoritmo, considere la opción de anulaciones imprevistas de reservas.

7.11. Cada alumno de una clase de licenciatura en Ciencias de la Computación tiene notas correspondientes a ocho asignaturas diferentes, pudiendo no tener calificación en alguna asignatura. A cada asignatura le corresponde un determinado coeficiente. Escribir un algoritmo que permita calcular la media de cada alumno.

Modificar el algoritmo para obtener las siguientes medias:

- General de la clase
- De la clase en cada asignatura
- Porcentaje de faltas (no presentado a examen)

7.12. Escribir un algoritmo que permita calcular el cuadrado de los 100 primeros números enteros y a continuación escribir una tabla que contenga dichos cuadrados.

7.13. Se dispone de N temperaturas almacenadas en un array. Se desea calcular su media y obtener el número de temperaturas mayores o iguales que la media.

7.14. Calcular la suma de todos los elementos de un vector de dimensión 100, así como su media aritmética.

7.15. Diseñar un algoritmo que calcule el mayor valor de una lista L de N elementos.

7.16. Dada una lista L de N elementos, diseñar un algoritmo que calcule de forma independiente la suma de los números pares y la suma de los números impares.

7.17. Escribir el algoritmo que permita escribir el contenido de una tabla de dos dimensiones (3 × 4).

7.18. Leer una matriz de 3 × 3.

7.19. Escribir un algoritmo que permita sumar el número de elementos positivos y el de negativos de una tabla T de n filas y m columnas.

7.20. Se dispone de las notas de 40 alumnos. Cada uno puede tener una o varias notas. Escribir un algoritmo que permita obtener la media de cada alumno y la media de la clase a partir de la entrada de las notas desde la computadora.

7.21. Una empresa tiene diez almacenes y necesita crear un algoritmo que lea las ventas mensuales de cada uno de ellos, calcular la media de ventas y obtener un listado de los almacenes cuyas ventas mensuales son superiores a la media.

7.22. Se dispone de una lista de cien números enteros. Calcular su valor máximo y el orden que ocupa en la tabla.

7.23. Un avión dispone de 180 asientos, de los cuales 20 son clase "ejecutiva", numerados de 1 al 20 y 160 asientos numerados 21 al 180 de clase "turista". Diseñar un algoritmo que permita hacer la reserva de asientos del avión y se detenga media hora antes de la salida, en cuyo momento se abrirá la lista de espera.

7.24. Calcular las medias de las estaturas de una clase. Deducir cuántos son más altos que la media y cuántos más bajos.

7.25. Las notas de un colegio se tienen en una matriz de 30 × 5 elementos (30, número de alumnos; 5, número de asignaturas). Se desea listar las notas de cada alumno y su media. Cada alumno tiene como mínimo dos asignaturas y como máximo cinco; aunque no todos necesariamente deben tener cinco materias.

7.26. Dado el nombre de una serie de estudiantes y las calificaciones obtenidas en un examen, calcular e imprimir la calificación media, así como cada calificación y la diferencia con la media.

7.27. Se introducen una serie de valores numéricos desde el teclado, siendo el valor final de entrada de datos o centinela –99. Se desea calcular e imprimir el número de valores leídos, la suma y media de los valores y una tabla que muestre cada valor leído y sus desviaciones de la media.

7.28. Se dispone de una lista de N nombres de alumnos. Escribir un algoritmo que solicite el nombre de un alumno y busque en la lista (*array*) si el nombre está en la lista.

Krasher Padres 1095564-2202172305

CAPÍTULO 8

Las cadenas de caracteres

8.1. Introducción
8.2. El juego de caracteres
8.3. Cadena de caracteres
8.4. Datos tipo carácter
8.5. Operaciones con cadenas
8.6. Otras funciones de cadenas
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Krasner Podres 1095584-2202172306

1095584-2202172306

INTRODUCCIÓN

Las computadoras normalmente sugieren operaciones aritméticas ejecutadas sobre datos numéricos. Sin embargo, ese concepto no es estadísticamente cierto, sino que, al contrario, hoy día es cada vez más frecuente el uso de las computadoras para procesar problemas de tipo esencialmente alfanuméricos o de tipo texto. En el capítulo 3 se estudió el concepto de tipo de datos carácter (**char**) y se definió un carácter como un símbolo del juego de caracteres de la computadora. Una **constante carácter** se definió como cualquier carácter encerrado entre separadores (apóstrofos o dobles comillas). Una secuencia finita de caracteres se denomina normalmente una **cadena** (*string*), y una **constante tipo cadena** es en una cadena encerrada entre apóstrofos o comillas dobles. El procesamiento de cadenas es el objetivo fundamental de este capítulo.

8.1. Introducción

Las computadoras nacieron para resolver problemas numéricos en cálculos científicos y matemáticos. Sin embargo, el paso de los años ha cambiado las aplicaciones y hoy día las computadoras no sólo se utilizan en cálculos numéricos, sino también para procesar datos de caracteres. En aplicaciones de gestión, la generación y actualización de listas de dirección, inventarios, etcétera, la información alfabética es fundamental. La edición de textos, traductores de lenguajes y base de datos son otras aplicaciones donde las cadenas de caracteres tienen gran utilidad.

En este capítulo se tratará el concepto de cadena de caracteres y su procesamiento, para ello se empleará una notación algorítmica similar a la utilizada hasta ahora. Una *cadena de caracteres* es una secuencia de cero o más símbolos, que incluyen letras del alfabeto, dígitos y caracteres especiales.

8.2. El juego de caracteres

Los lenguajes de programación utilizan *juegos de caracteres* "alfabeto" para comunicarse con las computadoras. Las primeras computadoras sólo utilizaban informaciones numéricas digitales mediante el código o alfabeto digital, y los primeros programas se escribieron en ese tipo de código, denominado *código máquina* —basado en dos dígitos, 0 y 1—, por ser inteligible directamente por la máquina (computadora). La enojosa tarea de programar en código máquina hizo que el alfabeto evolucionase y los lenguajes de programación comenzaran a utilizar códigos o juegos de caracteres similares al utilizado en los lenguajes humanos. Así, hoy día la mayoría de las computadoras trabajan con diferentes tipos de juegos de caracteres de los que se destacan el código ASCII y el EBCDIC.

De este modo, una computadora a través de los diferentes lenguajes de programación utiliza un juego o código de caracteres que será fácilmente interpretado y que puede ser programado por el usuario. Tres son los códigos más utilizados actualmente en computadoras, **ASCII** (American Standard Code for Information Interchange), **EBCDIC** (Extended Binary Coded Decimal Interchange Code) y **Unicode**.

El *código ASCII básico* utiliza 7 *bits* (dígitos binarios, 0, 1) para cada carácter a representar, lo que supone un total de 27 (128) caracteres distintos. El código ASCII ampliado utiliza 8 bits y, en ese caso, consta de 256 caracteres. Este código ha adquirido una gran popularidad, ya que es el estándar en todas las familias de computadoras personales.

El *código EBCDIC* utiliza 8 *bits* por carácter y, por consiguiente, consta de 256 caracteres distintos. Su notoriedad reside en ser el utilizado por la firma IBM (sin embargo, en las computadoras personales PC, XT, AT y PS/2 IBM ha seguido el código ASCII).

El *código universal Unicode* para aplicación en Internet y en gran número de alfabetos internacionales. En general, un carácter ocupará un *byte* de almacenamiento de memoria.

8.2.1. Código ASCII

El código ASCII se compone de los siguientes tipos de caracteres:

- *Alfabéticos* (`a, b, ..., z/A, B, ..., Z`).
- *Numéricos* (`0, 1, 2, 3, ..., 8, 9`).
- *Especiales* (`+, -, *, /, (, ), <, >, etcétera`).
- *De control* son caracteres no imprimibles y que realizan una serie de funciones relacionadas con la escritura, transmisión de datos, separador de archivos, etcétera, en realidad con los dispositivos de entrada/salida. Destacamos entre ellos:

DEL	*eliminar o borrar*
STX	*inicio de texto*
LF	*avance de línea*
FF	*avance de página*
CR	*retorno de carro*

Los caracteres del 128 al 255, pertenecientes en exclusiva al código ASCII ampliado, no suelen ser estándar y normalmente cada fabricante los utiliza para situar en ellos caracteres específicos de su máquina o de otros alfabetos, caracteres gráficos, etcétera. En la figura 8.2 se muestra el código ASCII de la familia de computadoras IBM PC y compatibles, donde se puede apreciar tanto el ASCII básico estándar como el ampliado.

Krasner Podres 1095584-2202172306

Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter
000	NUL	032	espacio	064	@	096	'
001	SOH	033	!	065	A	097	a
002	STX	034	"	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	$	068	D	100	d
005	ENQ	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	'	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	+	075	K	107	k
012	FF	044	,	076	L	108	l
013	CR	045	–	077	M	109	m
014	SO	046	.	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	:	090	Z	122	z
027	ESC	059	;	091	[	123	{
028	FS	060	<	092	\	124	|
029	GS	061	=	093	]	125	}
030	RS	062	>	094	↑	126	~
031	US	063	?	095	_	127	DEL

NOTA: Los 32 primeros caracteres y el último son caracteres de control; no son imprimibles.

Figura 8.1. Código ASCII básico.

Krasner Podnes 1095584-2202172308

8.2.2. Código EBCDIC

Este código es muy similar al ASCII, incluyendo también, además de los caracteres alfanuméricos y especiales, caracteres de control. Es propio de computadoras de IBM, con la excepción de los modelos PC, XT, AT y PS/2.

8.2.3. Código universal Unicode para Internet

Aunque ASCII es un código ampliamente utilizado para textos en inglés, es muy limitado, ya que un código de un byte sólo puede representar 256 caracteres diferentes ($2^8 = 256$). El lenguaje Java comenzó a utilizar la representación internacional *Unicode* más moderna y más amplia en juego de caracteres, ya que es un código de dos bytes (16 bits), que permiten hasta 65536 caracteres diferentes ($2^{16} = 65536$).

El código **Unicode** es un estándar internacional que define la representación de caracteres de una amplia gama de alfabetos. Tradicionalmente, como ya se ha comentado, los lenguajes de programación utilizaban el código ASCII cuyo juego de caracteres era 127 (o 256 para el código ASCII ampliado) que se almacenaban en 7 (o en 8) bits y que básicamente incluían aquellos caracteres que aparecían en el teclado estándar (QWERTY). Para los programadores que escriben en inglés estos caracteres son más o menos suficientes. Sin embargo, la aparición de Java y posteriormente C# como lenguajes universales requieren que éstos puedan ser utilizados en lenguajes internacionales como español, alemán, francés, chino, etcétera. Esta característica requiere de más de 256 caracteres diferentes. La representación *Unicode* que admite hasta 65536 resuelve estos problemas.

D	P	D	P	D	P	D	P	D	P	D	P	D	P	D	P
0		32		64	@	96		128	Ç	160	á	192	└	224	α
1	☺	33	!	65	A	97	a	129	ü	161	í	193	┴	225	β
2		34	"	66	B	98	b	130	é	162	ó	194	┬	226	Γ
3	♥	35	#	67	C	99	c	131	â	163	ú	195	├	227	π
4	♦	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	Σ
5	♣	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	σ
6	♠	38	&	70	F	102	f	134	å	166	ª	198	╞	230	µ
7	•	39	´	71	G	103	g	135	ç	167	º	199		231	γ
8	◘	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	φ
9	○	41	)	73	I	105	i	137	ë	169	⌐	201	╔	233	θ
10	■	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Ω
11	♂	43	+	75	K	107	k	139	ï	171	½	203	╦	235	δ
12	♀	44	,	76	L	108	l	140	î	172	¼	204	╠	236	∞
13	♪	45	–	77	M	109	m	141	ì	173	¡	205	═	237	∅
14	♫	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	∈
15	☼	47	/	79	O	111	o	143	Å	175	»	207	╧	239	∩
16	►	48	0	80	P	112	p	144	É	176		208	╨	240	≡
17	◄	49	I	81	Q	113	q	145	æ	177	▒	209	╤	241	±
18	↕	50	2	82	R	114	r	146	Æ	178	▓	210	╥	242	≥
19	‼	51	3	83	S	115	s	147	ô	179	│	211	╙	243	≤
20	¶	52	4	84	T	116	t	148	ö	180	┤	212	╘	244	⌠
21	§	53	5	85	U	117	u	149	ò	181	╡	213	╒	245	⌡
22	▬	54	6	86	V	118	v	150	û	182	╢	214	╓	246	÷
23	↨	55	7	87	W	119	w	151	ù	183	╖	215	╫	247	≈
24	↑	56	8	88	X	120	x	152	ÿ	184	╕	216	╪	248	°
25	↓	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	•
26	→	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
27	←	59	;	91	[	123	{	155	¢	187	╗	219	▮	251	√
28	∟	60	<	92	\	124	|	156	£	188	╝	220	▄	252	ⁿ
29	↔	61	=	93	]	125	}	157	¥	189	╜	221	▌	253	²
30	▲	62	>	94	^	126	~	158	Pt	190	╛	222	▐	254	■
31	▼	63	?	95	_	127	⌂	159	ƒ	191	┐	223	▀	255	

D: Código decimal.
P: Escritura del carácter correspondiente al código en la pantalla.

Figura 8.2. Código ASCII de la computadora IBM PC.

D	C	D	C	D	C	D	C	D	C	D	C	D	C	D	C
0	NUL	21	NL	43	CU2	79	,	124	@	150	o	195	C	227	T
1	SOH	22	BS	45	ENQ	80	&	125	´	151	p	196	D	228	U
2	STX	23	IL	46	ACK	90	!	126	=	152	q	197	E	229	V
3	ETX	24	CAN	47	BEL	91	$	127	"	153	r	198	F	230	W
4	PF	25	EM	50	SYN	92	*	129	a	155	}	199	G	231	X
5	HT	26	CC	52	PN	93	)	130	b	161	~	200	H	232	Y
6	LC	27	CU1	53	RS	94	;	131	c	162	s	201	I	233	Z
7	DEL	28	IFS	54	UC	95	¬	132	d	163	t	208	}	240	0
10	SMM	29	IGS	55	EOT	96	–	133	e	164	u	209	J	241	1
11	VT	30	IRS	59	CU3	97	/	134	f	165	v	210	K	242	2
12	FF	31	IUS	60	DC4	106	¦	135	g	166	w	211	L	243	3
13	CR	32	DS	61	NAK	107	,	136	h	167	x	212	M	244	4
14	SO	33	SOS	63	SUB	108	%	137	I	168	y	213	N	245	5
15	SI	34	FS	64	SP	109	_	139	{	169	z	214	O	246	6
16	DLE	36	BYP	74	c	110	>	145	J	173	[	215	P	247	7
17	DC1	37	LF	75	.	111	?	146	k	189	]	216	Q	248	8
18	DC2	38	ETB	76	<	121	`	147	l	192	{	217	R	249	9
19	DC3	39	ESC	77	(	122	:	148	m	193	A	224	\	250	I
20	RES	40	SM	78	+	123	#	149	n	194	B	226	S		

D: Código decimal.
P: Escritura del carácter correspondiente al código en la pantalla.

Figura 8.3. Código EBCDIC.

Krasher Padres 1095564-2202172309

En consecuencia, los identificadores en Java y C# deben comenzar con una letra Java o C#, que es cualquier carácter Unicode que no represente un dígito o un carácter de puntuación.

Las letras en inglés, así como los dígitos decimales y los signos de puntuación en inglés, se asignan a los códigos que son los mismos que en el código ASCII. Puede consultar los caracteres Unicode en el sitio Web oficial del consorcio **Unicode**:

```
http://www.unicode.org
```

8.2.4. Secuencias de escape

Una **secuencia de escape** es un medio para representar caracteres que no se pueden escribir desde el teclado y, por consiguiente, utilizarlos directamente en un editor. Una secuencia de escape consta de dos partes: el *carácter escape* y un *valor de traducción*; el primero es un símbolo que indica al compilador Java o C (por ejemplo) que ha de traducir el siguiente carácter de un modo especial. En Java, como en lenguaje C, este carácter de escape especial es la barra inclinada inversa (\).

Si la barra inclinada marca el principio de una secuencia de escape, ¿qué se puede utilizar para el valor de la traducción? La parte de la secuencia de escape que sigue al carácter escape y, tal vez, el valor de traducción más fácil para utilizar es un código de carácter Unicode; estos valores deben estar especificados como un número hexadecimal de cuatro dígitos precedido por una letra u. Los literales de caracteres Java o C# se deben encerrar entre comillas simples:

Sintaxis '\uxxxx'
Ejemplos '\u0344' '\u2122'

En programas escritos en cualquier lenguaje (en particular en Java o en C#) se pueden utilizar las secuencias de escape Unicode en cualquier parte donde algún tipo de carácter pueda aparecer: en literales "carácter", en literales "cadenas" o incluso en identificadores.

Todos los lenguajes de programación (C, C++, Java, etcétera) permiten especificar el carácter de escape para especificar otros tipos de caracteres especiales; entre ellos están los "caracteres invisibles" que se han utilizado tradicionalmente para controlar operaciones de computadora (a veces se les conoce también como "caracteres de control") así como simples comillas, comillas dobles y el propio carácter de escape. Así, para escribir una comilla simple como una literal carácter, se escribe '\''. La tabla 8.1 proporciona las secuencias de escape que el lenguaje Java reconoce.

Tabla 8.1. Secuencias de escape en Java.

Secuencia	Significado
\b	Retroceso (\u0008)
\t	Tabulación (\U0009)
\n	Nueva línea (\u000A)
\f	Avance de página (\u000C)
\r	Retorno de carro (\u000D)
\"	Dobles comillas (\u0022)
\'	Comillas simples (\u0027)
\\	Barra inclinada inversa (\u005C)
\\ddd	Cualquier carácter especificado por dígitos octales *ddd*

8.3. Cadena de caracteres

Una *cadena* (*string*) de caracteres es un conjunto de caracteres –incluido el blanco– que se almacenan en un área contigua de la memoria. Pueden ser entradas o salidas a/desde un terminal.

Krasner Podres 1095584-2202172312

La *longitud* de una cadena es el número de caracteres que contiene. La cadena que no contiene ningún carácter se le denomina *cadena vacía o nula*, y su longitud es cero; no se debe confundir con una cadena compuesta sólo de blancos —espacios en blanco—, ya que ésta tendrá como longitud el número de blancos de la misma.

La representación de las cadenas suele ser con comillas simples o dobles. En nuestro libro utilizaremos las comillas simples por ser esa notación la más antigua utilizada en diversos lenguajes como Pascal, Fortran, etcétera, aunque hoy día los lenguajes modernos, tales como C, C++, Java y C#, utilizan las comillas dobles.

Notaciones de cadenas

Pascal, Fortran, UPSAM	`'Cartagena de Indias'`
C, C++, Java, C#	`"Cartagena de Indias"`

EJEMPLO 8.1

```
'12 de octubre de 1492'
'Por fin llegaste'
'            '
'AMÉRICA ES GRANDE'
```

Las cadenas pueden contener cualquier carácter válido del código aceptado por el lenguaje y la computadora; el blanco es uno de los caracteres más utilizado; si se le quiere representar de modo especial en la escritura en papel, se emplea alguno de los siguientes símbolos:

_ ␣ □ ∪

Por nuestra parte utilizaremos _, dejando libertad al lector para usar el que mejor convenga a su estilo de programación. Las cadenas anteriores tienen longitudes respectivas de 21, 16, 3 y 17.

Una *subcadena* es una cadena de caracteres que ha sido extraída de otra de mayor longitud.

`'12 de'`	*es una subcadena de*	`'12 de octubre'`
`'Java'`	*es una subcadena de*	`'lenguaje Java'`
`'CHE'`	*es una subcadena de*	`'CARCHELEJO'`

Reglas de sintaxis en lenguajes de programación

C++	Una cadena es un *array* de caracteres terminado con el carácter nulo, cuya representación es la secuencia de escape '0' y su nombre es NULL (nulo).
C#	Las cadenas son objetos del tipo incorporado *String*. En realidad, *String* es una clase que proporciona funcionalidades de manipulación de cadenas y en particular construcción de cadenas.
Java...	Las cadenas son objetos del tipo *String*. *String* es una clase en Java y una vez que los objetos cadena se crean, el contenido no se puede modificar, aunque pueden ser construidas todas las cadenas que se deseen.

EJEMPLO 8.2

Cadena 'Carchelejo' representada en lenguaje C++.

C	A	R	C	H	E	L	E	J	O	\0

Krasner Podres 1095584-2202172313

8.4. Datos tipo carácter

En el capítulo 3 se analizaron los diferentes tipos de datos y entre ellos existía el dato tipo *carácter (char)* que se incorpora en diferentes lenguajes de programación, bien con este nombre o bien como datos tipo cadena. Así pues, en esta sección trataremos las constantes y las variables tipo carácter o cadena.

8.4.1. Constantes

Una constante tipo carácter es un carácter encerrado entre comillas y una constante de tipo cadena es un conjunto de caracteres válidos encerrados entre comillas —apóstrofos— para evitar confundirlos con nombres de variables, operadores, enteros, etcétera. Si se desea escribir un carácter comilla, se debe escribir duplicado. Como se ha comentado anteriormente, existen lenguajes —Basic, C, C++, Java, etcétera, por ejemplo— que encierran las cadenas entre comillas dobles. Nuestros algoritmos sólo tendrán una, por seguir razones históricas y por compatibilidad con versiones anteriores del lenguaje UPSAM.

```
'Carchelejo es un pueblo de Jaen'
```

es una constante de tipo cadena, de una longitud fija igual a 31.

```
'¿'
```

es una constante de tipo carácter.

Krasner Podres 1095564-2202172314

8.4.2. Variables

Una *variable de cadena* o *tipo carácter* es una variable cuyo valor es una cadena de caracteres.

Las variables de tipo carácter o cadena se deben declarar en el algoritmo y según el lenguaje tendrán una notación u otra. Nosotros, al igual que muchos lenguajes, las declararemos en la tabla o bloque de declaración de variables.

```
var
   carácter : A, B
   cadena : NOMBRE, DIRECCIÓN
```

Atendiendo a la declaración de la longitud, las variables se dividen en *estáticas, semiestáticas* y *dinámicas*.

Variables estáticas son aquellas en las que su longitud se define antes de ejecutar el programa y no puede cambiarse a lo largo de éste.

Fortran: `CHARACTER A1 * 10, A2 * 15`

las variables `A1` y `A2` se declaran con longitudes 10 y 15, respectivamente.

```
Pascal:          var NOMBRE: PACKED ARRAY [1..30] OF CHAR
Turbo Pascal:    var NOMBRE: array[1..30] of char          o bien
                 var NOMBRE:STRING[30]
```

En Pascal, una variable de tipo carácter —*char*— sólo puede almacenar un carácter y, por consiguiente, una cadena de caracteres debe representarse mediante un *array* de caracteres. En el ejemplo, `NOMBRE` se declara como una cadena de 30 caracteres (en este caso, `NOMBRE[1]` será el primer carácter de la cadena, `NOMBRE[2]` será el segundo carácter de la cadena, y así en lo sucesivo).

Turbo Pascal admite también tratamiento de cadenas semiestáticas `(STRING)` como dato.

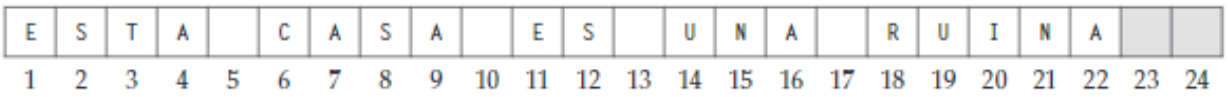
Variables semiestáticas son aquellas cuya longitud puede variar durante la ejecución del programa, pero sin sobrepasar un límite máximo declarado al principio.

Variables dinámicas son aquellas cuya longitud puede variar sin limitación dentro del programa. El lenguaje Snobol es típico de variables dinámicas.

La representación de las diferentes variables de cadena en memoria utiliza un método de almacenamiento diferente.

Cadenas de longitud fija

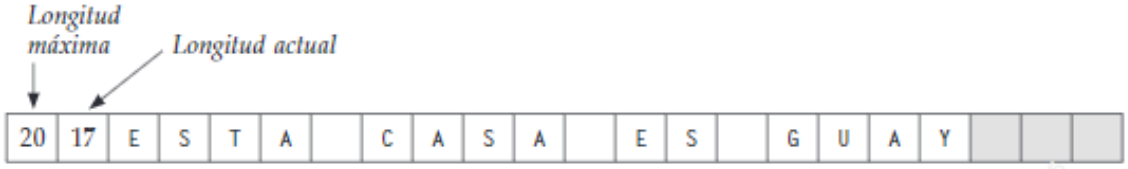
Se consideran vectores de la longitud declarada, con blancos a izquierda o derecha si la cadena no tiene la longitud declarada. Así, la cadena siguiente:



se declaró con una dimensión de 24 caracteres y los dos últimos se rellenan con blancos.

Cadenas de longitud variable con un máximo

Se considera un puntero (en el capítulo 12 ampliaremos este concepto) con dos campos que contienen la longitud máxima y la longitud actual.



Cadenas de longitud indefinida

Se representan mediante listas enlazadas, que son listas que se unen mediante puntero.

Estas listas contienen elementos con caracteres empaquetados –2/elemento– y enlazados cada uno con el siguiente por un puntero (la cadena de caracteres es `'MADONA'`).

8.4.3. Instrucciones básicas con cadenas

Las instrucciones básicas: *asignar* y *entrada/salida (leer/escribir)* se realizan de un modo similar al tratamiento de dichas instrucciones con datos numéricos.

Asignación

Si la variable `NOMBRE` se ha declarado como tipo cadena

```
var cadena : NOMBRE
```

la instrucción de asignación debe contener en el lado derecho de la asignación una constante tipo cadena o bien otra variable del mismo tipo. Así,

```
NOMBRE ←'Luis Hermenegildo'
```

significa que la variable `NOMBRE` toma por valor la cadena `'Luis Hermenegildo'`.

Krasner Podres 1095584-2202172315

Entrada/Salida

La entrada/salida desde una terminal se puede realizar en modo carácter; para ello bastará asignar —a través del correspondiente dispositivo— una cadena de caracteres a una variable tipo cadena. Así, por ejemplo, si `A`, `B`, `C` y `D` se han declarado como variables tipo cadena

```
var cadena : A, B, C, D
```

las instrucciones:

```
leer(A, B)
escribir(C, D)
```

asignarán a `A` y `B` las cadenas introducidas por el teclado y visualizarán o imprimirán en el dispositivo de salida las cadenas que representan las variables `C` y `D`.

8.5. Operaciones con cadenas

El tratamiento de cadenas es un tema importante, debido esencialmente a la gran cantidad de información que se almacena en ellas. Según el tipo de lenguaje de programación elegido se tendrá mayor o menor facilidad para la realización de operaciones. Así, por ejemplo, C tiene grandes posibilidades, Fortran sólo operaciones elementales, y Pascal, dependiendo del compilador, soporta procedimientos y funciones predefinidas o es preciso definirlos por el usuario con la natural complejidad que suponga el diseño del algoritmo correspondiente. Todos los lenguajes orientados a objetos como C++, C# y Java, merced a la clase `String` soportan una gran gama de funciones de manipulación de cadenas. En cualquier caso, las operaciones con cadena más usuales son:

- Cálculo de la longitud.
- Comparación.
- Concatenación.
- Extracción de *subcadenas*.
- Búsqueda de información.

Krasner Podres 1095584-2202172316

8.5.1. Cálculo de la longitud de una cadena

La longitud de una cadena, como ya se ha comentado, es el número de caracteres de la cadena. Así:

```
'Don Quijote de la Mancha'
```

tiene veinticuatro caracteres.

La operación de determinación de la longitud de una cadena se representará por la función **longitud**, cuyo formato es:

```
longitud (cadena)
```

La función `longitud` tiene como argumento una cadena, pero su resultado es un valor numérico entero:

```
longitud('Don Quijote de la Mancha')   proporciona 24
longitud('␣␣␣')                        cadena de tres blancos proporciona 3
longitud('␣␣␣Mortadelo')               cadena 'Mortadelo' rellenada de blancos a la izquierda
                                       para tener longitud 12
```

En consecuencia, la función `longitud` se puede considerar un dato tipo entero y, por consiguiente, puede ser un operando dentro de expresiones aritméticas.

```
4 + 5 + longitud('DEMO') = 4+5+4 = 13
```

8.5.2. Comparación

La *comparación* de cadenas (igualdad y desigualdad) es una operación muy importante, sobre todo en la clasificación de datos tipo carácter que se utiliza con mucha frecuencia en aplicaciones de proceso de datos (clasificaciones de listas, tratamiento de textos, etcétera).

Los criterios de comparación se basan en el orden numérico del código o juego de caracteres que admite la computadora o el propio lenguaje de programación. En nuestro lenguaje algorítmico utilizaremos el código ASCII como código numérico de referencia. Así,

- El carácter 'A' *será <* el carácter 'C'
  (código 65) *(código 67)*
- El carácter '8' *será <* el carácter 'i'
  (código 56) *(código 105)*

En la comparación de cadenas se pueden considerar dos operaciones más elementales: *igualdad* y *desigualdad*.

Igualdad

Dos cadenas *a* y *b* de longitudes *m* y *n* son iguales si:

- El número de caracteres de *a* y *b* son los mismos *(m = n)*.
- Cada carácter de *a* es igual a su correspondiente de *b* si $a = a_1a_2...a_n$ y $b = b_1b_2...b_n$ se debe verificar que $a_i = b_i$ para todo *i* en el rango $1 <= i <= n$.

Así:
```
'EMILIO' = 'EMILIO'    es una expresión verdadera
'EMILIO' = 'EMILIA'    es una expresión falsa
'EMILIO' = 'EMILIO'    es una expresión falsa; contiene un blanco final y, por consiguiente, las
                       longitudes no son iguales.
```

Desigualdad

Los criterios para comprobar la desigualdad de cadena son utilizados por los operadores de relación <, <=, >=, < > y se ajustan a una comparación sucesiva de caracteres correspondientes en ambas cadenas hasta conseguir dos caracteres diferentes. De este modo, se pueden conseguir clasificaciones alfanuméricas

```
'GARCIA' < 'GOMEZ'
```

ya que las comparaciones sucesivas de caracteres es:

```
G-A-R-C-I-A        G = G, A < O, ...
G-O-M-E-Z
```

una vez que se encuentra una desigualdad, no es preciso continuar; como se observa, las cadenas no tienen por qué tener la misma longitud para ser comparadas.

EJEMPLO 8.3

En las sucesivas comparaciones se puede apreciar una amplia gama de posibles casos.

'LUIS'	<	'LUISITO'	*verdadera*
'ANA'	<	'MARTA'	*verdadera*
'TOMAS'	<	'LUIS'	*falsa*
'BARTOLO'	<=	'BARTOLOME'	*verdadera*
'CARMONA'	>	'MADRID'	*falsa*
'LUIS'	>	'LUIS'	*verdadera*

Se puede observar de los casos anteriores que la presencia de cualquier carácter —incluso el blanco—, se considera mayor siempre que la ausencia. Por eso, 'LUIS' es mayor que 'LUIS'.

Krasner Padres 1095564-2202172321

8.5.3. Concatenación

La concatenación es la operación de reunir varias cadenas de caracteres en una sola, pero conservando el orden de los caracteres de cada una de ellas.

El símbolo que representa la concatenación varía de unos lenguajes a otros. Los más utilizados son:

```
+   //      &       o
```

En nuestro libro utilizaremos & y en ocasiones +. El símbolo & evita confusiones con el operador suma. Las cadenas para concatenarse pueden ser constantes o variables.

```
'MIGUEL'&'DE'&'CERVANTES' == 'MIGUELDECERVANTES'
```

Puede comprobar que las cadenas, en realidad, se *pegan* unas al lado de las otras; por ello, si al concatenar frases desea dejar blancos entre ellas, deberá indicarlos expresamente en alguna de las cadenas. Así, las operaciones

```
'MIGUEL '&' DE '&' CERVANTES
'MIGUEL '&' DE '&' CERVANTES
```

producen el mismo resultado:

```
'MIGUEL DE CERVANTES'
```

lo que significa que la *propiedad asociativa* se cumple en la operación de concatenación.

> El operador de concatenación (+, &) actúa como un operador aritmético.

EJEMPLO 8.4

Es posible concatenar variables de cadena.

```
var cadena : A, B, C
A&B&C equivale a A&(B&C)
```

La asignación de constantes tipo cadena a variables tipo cadena puede también realizarse con expresiones concatenadas.

EJEMPLO 8.5

Las variables A, B son de tipo cadena.

```
var cadena : A, B
A ←'FUNDAMENTOS'
B ←'DE PROGRAMACION'
```

La variable C puede recibir como valor:

```
C ←A+' '+B
```

que produce un resultado de:

```
C = 'FUNDAMENTOS DE PROGRAMACION'
```

Krasner Podres 1095584-2202172331

Concatenación en Java

El lenguaje Java soporta la concatenación de cadenas mediante el operador + que actúa sobrecargado. Así, suponiendo que la cadena c1 contiene "Fiestas de moros" y la cadena c2 contiene "y cristianos", la cadena c1 + c2 almacenará "Fiestas de moros y cristianos".

8.5.4. Subcadenas

Otra operación —función— importante de las cadenas es aquella que permite la extracción de una parte específica de una cadena: *subcadena*. La operación *subcadena* se representa en dos formatos por:

```
subcadena (cadena, inicio, longitud)
```

- *Cadena* es la cadena de la que debe extraerse una subcadena.
- *Inicio* es un número o expresión numérica entera que corresponde a la posición inicial de la subcadena.
- *Longitud* es la longitud de la subcadena.

```
subcadena (cadena, inicio)
```

En este caso, la subcadena comienza en *inicio* y termina en el final de la cadena.

Krasher Padres 1095564-2202172332

1095564-2202172332

EJEMPLOS

subcadena	('abcdef', 2, 4)	*equivale a*	'bcde'
subcadena	('abcdef', 6, 1)	*equivale a*	'f'
subcadena	('abcdef', 3)	*equivale a*	'cdef'
subcadena	('abcdef', 3, 4)	*equivale a*	'cdef'

```
    longitud = 5 caracteres
            ⌒⌒⌒
subcadena ('12 DE OCTUBRE', 4, 5) = DE OC
              ↑
            posición 4
```

Es posible realizar operaciones de concatenación con subcadenas.

```
subcadena ('PATO DONALD',1,4) + subcadena ('ESTA TIERRA',5,4)
```

equivale a la cadena 'PATO TIE'.

La aplicación de la función a una subcadena,

```
subcadena (cadena, inicio, fin)
```

puede producir los siguientes resultados:

1. Si *fin* no existe, entonces la subcadena comienza en el mismo carácter inicio y termina con el último carácter.
2. Si *fin* <= 0, el resultado es una cadena vacía.
3. Si *inicio* > *longitud (cadena)*, la subcadena resultante será vacía.

   ```
   subcadena ('MORTIMER', 9, 2)
   ```

   produce una cadena vacía.

4. Si *inicio* <= 0, el resultado es también una cadena vacía.
   subcadena `('valdez', 0, 4)` *y subcadena* `('valdez', 8)`
   proporcionan cadenas nulas.

8.5.5. Búsqueda

Una operación frecuente a realizar con cadenas es localizar si una determinada cadena forma parte de otra cadena más grande o buscar la posición en que aparece un determinado carácter o secuencia de caracteres de un texto.

Estos problemas pueden resolverse con las funciones de cadena estudiadas hasta ahora, pero será necesario diseñar los algoritmos correspondientes. Esta función suele ser interna en algunos lenguajes y la definiremos por **`indice`** o **`posicion`**, y su formato es:

```
indice (cadena, subcadena)
```

o bien:

```
posicion (cadena, subcadena)
```

donde *subcadena* es el texto que se trata de localizar.

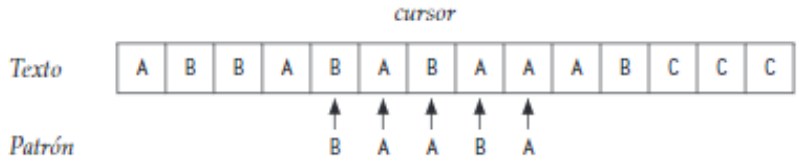
El resultado de la función es un valor entero:

- Igual a $P >= 1$, donde P indica la posición del primer carácter de la primera coincidencia de subcadena en cadena.
- Igual a cero, si subcadena es una cadena vacía o no aparece en la cadena.

Así, suponiendo la `cadena C = 'LA CAPITAL ES MADRID'`

```
indice (C, 'CAP')        toma un valor 4
indice (C, ' ES ')       toma un valor 11
indice (C, 'PADRID')     toma un valor 0
```

La función **`indice`** en su forma más general realiza la operación que se denomina ***coincidencia de patrones*** *(patter-matching)*. Esta operación busca una cadena patrón o modelo dentro de una cadena de texto.



Esta operación utiliza un cursor o puntero en la cadena de texto original y va comprobando los sucesivos valores de ambas cadenas: si son distintos, produce un 0, y si no proporciona la posición del primer carácter coincidente.

```
indice  ('ABCDE', 'F')          produce 0
indice  ('ABXYZCDEF', 'XYZ')    produce 3
```

La función **`indice (posicion)`** al tomar también un valor numérico entero se puede utilizar en expresiones aritméticas o en instrucciones de asignación a variables numéricas.

```
P ← indice (C, 'F')
```

8.6. Otras funciones de cadenas

Existen otras funciones de cadena internas al lenguaje o definidas por el usuario, que suelen ser de utilidad en programación y cuyo conocimiento es importante que conozca el lector:

Krasher Podres 1095584-2202172333

- *Insertar* cadenas.
- *Borrar* cadenas.
- *Cambiar* cadenas.
- *Convertir* cadenas en números y viceversa.

8.6.1. Insertar

Si se desea insertar una cadena C dentro de un texto o cadena más grande, se debe indicar la posición. El formato de la función **insertar** es:

```
insertar (t, p, s)
```

- *t* texto o cadena *donde* se va a insertar.
- *p* posición *a partir de la cual* se va a insertar.
- *s* *subcadena* que se va a *insertar*.

```
insertar ('ABCDEFGHI', 4, 'XXX') = 'ABCXXXDEFGHI'
insertar ('MARÍA O', 7, 'DE LA') = 'MARÍA DE LA O'
```

Kristine Podres 1095584-2202172335

Algoritmo de inserción

Si su lenguaje no posee definida esta función, se puede implementar con el siguiente algoritmo:

```
inicio
   insertar(t, p, s) = subcadena(t,1,p-1)& S &
      subcadena(t, p, longitud(t)-p+1)
fin
```

1095584220217233S

Veámoslo con un ejemplo: **insertar** ('ABCDEFGHI' 4, 'XXX')

donde t = 'ABCDEFGHI' y S = 'XXX' p = 4

```
subcadena (t,1,p-1) = subcadena (t,1,3) = ABC
subcadena (t, p, longitud(t)-p+1) = subcadena (t,4,9-4+1) =
subcadena (t,4,6) = DEFGHI
```

por consiguiente:

```
insertar ('ABCDEFGHI',4'XXX')= 'ABC'+'XXX'+'DEFGHI'='ABCXXXDEFGHI'
```

8.6.2. Borrar

Si se desea eliminar una subcadena que comienza en la posición *p* y tiene una longitud *l* se tiene la función **borrar**.

```
borrar (t, p, l)
```

- *t* texto o cadena de donde se va a eliminar una subcadena,
- *p* posición a partir de la cual se va a borrar (eliminar),
- *l* longitud de la subcadena a eliminar,

```
borrar ('supercalifragilistico', 6, 4) = 'superfragilistico'
borrar ('supercalifragilistico', 3, 10) = 'sugilistico'
```

Algoritmo borrar

Si no se posee la función estándar **borrar**, será preciso definirla. Ello se consigue con el algoritmo:

```
inicio
   borrar (t, p, 1) = subcadena (t,1,p-1) &
      subcadena (t, p+1, longitud(t)-p-1+1)
fin
```

8.6.3. Cambiar

La operación insertar trata de sustituir en un texto *t* la primera ocurrencia de una subcadena *S1* por otra *S2*. Éste es el caso frecuente en los programas de tratamiento de textos, donde a veces es necesario sustituir una palabra cualquiera por otra (... en el archivo "DEMO" sustituir la palabra "ordenador" por "computadora"), acomodando las posibles longitudes diferentes. La función que realiza la operación de insertar tiene el formato:

cambiar *(t, S1, S2)*

- *t* texto donde se realizarán los cambios.
- *S1* subcadena a sustituir.
- *S2* subcadena nueva.

```
cambiar ('ABCDEFGHIJ', 'DE', 'XXX') = 'ABCXXXFGHIJ'
```

Si la subcadena S1 no coincide exactamente con una subcadena de t, no se produce ningún cambio y el texto o cadena original no se modifica.

```
cambiar ('ABCDEFGHIJK', 'ZY', 'XXX') = 'ABCDEFGHIJK'
```

Algoritmo cambio

Si no se dispone de esta función como estándar, es posible definir un algoritmo haciendo uso de las funciones analizadas.

```
cambiar (t, S1, S2)
```

El algoritmo se realiza llamando a las funciones **indice**, **borrar** e **insertar**.

```
procedimiento cambiar(t, S1, S2)
inicio
   j ← indice(t, S1)
   t ← borrar(t, j, longitud(S1))
   insertar(t, j, S2)
fin
```

La primera instrucción, j ← **indice**(s, S1), calcula la posición donde se debe comenzar la inserción, que es, a su vez, el primer elemento de la subcadena S1.

La segunda instrucción:

```
t ←borrar(t, j, longitud(S1))
```

borra la subcadena S1 y la nueva cadena se asigna a la variable de cadena t.

La tercera instrucción inserta en la nueva cadena t —original sin la cadena S1— la subcadena S2 a partir del carácter de posición j, como se había previsto.

8.6.4. Conversión de cadenas/números

Existen funciones o procedimientos en los lenguajes de programación (**val** y **str** en Basic, **val** y **str** en Turbo Pascal) que permiten convertir un número en una cadena y viceversa.

Krasner Podres 1095584-2202172338

En nuestro algoritmo los denotaremos por **valor** y **cad**.

`valor (cadena)`	*convierte la cadena en un número; siempre que la cadena fuese de dígitos numéricos*
`cad (valor)`	*convierte un valor numérico en una cadena*

EJEMPLOS

```
valor ('12345') = 12345
cad (12345) = '12345'
```

Otras funciones importantes relacionadas con la conversión de caracteres en números y de números en caracteres son:

`código (un_carácter)`	*devuelve el código ASCII de un carácter*
`car (un_código)`	*devuelve el carácter asociado en un código ASCII*

Krasner Padres 1095584-2202172339

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

8.1. *Se desea eliminar los blancos de una frase dada terminada en un punto. Se supone que es posible leer los caracteres de la frase de uno en uno.*

Solución

Análisis

Para poder efectuar la lectura de la frase, almacena ésta en un array de caracteres `(F)` —esto es posible en lenguajes como Pascal; en Basic sería preciso recurrir a enojosas tareas de operaciones con funciones de cadenas `MID$`, `LEFT$` o `RIG- HT$`—, de modo que `F[i]` contiene el carácter *i*-ésimo de la frase dada. Construiremos una nueva frase sin blancos en otro array `G`.

Algoritmo

Los pasos a dar para la realización del algoritmo son:

- Inicializar contador de letras de la nueva frase `G`.
- Leer el primer carácter.
- Repetir.

  Si el primer carácter no es en blanco, entonces
  escribir en el lugar siguiente del array G,
  leer carácter siguiente de la frase dada.
  Hasta que el último carácter se encuentre.
- Escribir la nueva frase —`G`— ya sin blancos.

Tabla de variables

`F` array de caracteres de la frase dada.
`G` array de caracteres de la nueva frase.

I contador del array F.
J contador del array G.

Pseudocódigo

```
algoritmo blanco
inicio
   I←1
   J←0
   F[i] ←leercar() {leercar es una función que permite la lectura de un carácter}
   repetir
      si F[I] <> ' ' entonces
         J ←J+1
         G[I] ←F[I]
      fin_si
      I ←I+1
      F[i] ←leercar()
   hasta_que F[I]='.'
   //escritura de la nueva frase G
   desde I ←1 hasta J hacer
      escribir(G[I])    //no avanzar linea
   fin_desde
fin
```

8.2. *Leer un carácter y deducir si está situado antes o después de la letra "m" en orden alfabético.*

Solución

Análisis

La comparación de datos de tipo carácter se realiza mediante los códigos numéricos ASCII, de modo que una letra estará situada antes o después de ésta si su código ASCII es menor o mayor. La propia computadora se encarga de realizar la comparación de datos tipo carácter de acuerdo con el código ASCII, siempre que los datos a comparar sean de tipo carácter. Por ello se deben declarar de tipo carácter las variables que representan las comparaciones.

```
Variables C: carácter
```

Pseudocódigo

```
algoritmo carácter
var
   carácter : C
inicio
   leer(C)
   si C < 'M' entonces
      escribir(C, 'esta antes que M en orden alfabético')
   si_no
      escribir(C, 'esta después que M en orden alfabético')
   fin_si
fin
```

8.3. *Leer los caracteres y deducir si están en orden alfabético.*

Solución

Tabla de variables

```
CAR1, CAR2: carácter
```

Krasnar Podres 1095584-2202172340

Pseudocódigo

```
algoritmo comparación
var
   carácter : CAR1, CAR2
inicio
   leer(CAR1, CAR2)
   si CAR1 <= CAR2 entonces
      escribir('en orden')
   si_no
      escribir('desordenados')
   fin_si
fin
```

8.4. *Leer una letra de un texto. Deducir si está o no comprendida entre las letras mayúsculas I-M inclusive.*

Solución

Variables

```
LETRA: carácter.
```

Pseudocódigo

```
algoritmo
var
   carácter : LETRA
inicio
   leer(LETRA)
   si (LETRA > = 'I') y (LETRA =< 'M') entonces
      escribir('está comprendida')
   si_no
      escribir('no está comprendida')
   fin_si
fin
```

8.5. *Contar el número de letras "i" de una frase terminada en un punto. Se supone que las letras pueden leerse independientemente.*

Solución

En este algoritmo el contador de letras sólo se incrementa cuando se encuentran las letras "i" buscadas.

Pseudocódigo

```
algoritmo letras_i
var
   entero : N
   carácter : LETRA
inicio
   N ← 0
   repetir
      LETRA ← leercar()
      si LETRA = 'i' entonces
         N ← N+1
      fin_si
   hasta_que LETRA = '.'
   escribir('La frase tiene', N, 'letras i')
fin
```

Krasnar Padres 1095584-2202172341

8.6. *Contar el número de vocales de una frase terminada en un punto.*

Solución

Pseudocódigo

```
algoritmo vocales
var
   entero : NÚMVOCALES
   carácter : C
inicio
   repetir
      C ← leercar()          {la función leercar permite la lectura de caracteres independientes}
      si C = 'a' o C = 'e' o C = 'i' o C = 'o' o C = 'u' entonces
         NÚMVOCALES ← NÚMVOCALES+1
      fin_si
   hasta_que C = '.'
   escribir('El número de vocales es =', NÚMVOCALES)
fin
```

8.7. *Se desea contar el número de letras "a" y el número de letras "b" de una frase terminada en un punto. Se supone que es posible leer los caracteres independientemente.*

Solución

Método 1

```
algoritmo letras_a_b
var
   entero : NA, NB
   carácter : C
inicio
   NA ←  0
   NB ←  0
   repetir
      C ← leercar()
      si C = 'a' entonces
         NA ← NA+1
      fin_si
      si C = 'b' entonces
         NB ← NB+1
      fin_si
   hasta_que C = '.'
   escribir('Letras a =', NA, 'Letras b=', NB)
fin
```

Método 2

```
algoritmo letras_a_b
var
   entero : NA, NB
   carácter : C
inicio
   NA ←  0
   NB ←  0
   repetir
      C ← leercar()
```

Krasher Padres 1095564-2202172343

```
            si C = 'a' entonces
               NA ← NA+1
         si_no
            si C = 'b' entonces
               NB ← NB+1
            fin_si
         fin_si
      hasta_que C = '.'
fin
```

Método 3

```
algoritmo letras_a_b
var
   entero : NA, NB
   carácter : C
inicio
   NA ←  0
   NB ←  0
   repetir
      C ← leercar()
      según_sea C hacer
         'a': NA ← NA+1
         'b': NB ← NB+1
      fin_según
   hasta_que C = '.'
fin
```

8.8. *Leer cien caracteres de un texto y contar el número de letras "b".*

Solución

Tabla de variables

```
entero : I, NE
carácter : C
```

Pseudocódigo

```
algoritmo letras_b
var
   entero : I, NE
   carácter : C
inicio
   NE ← 0
   desde I ← 1 hasta 100 hacer
      C ← leercar()
      si C = 'b' entonces
         NE ← NE+1
      fin_si
   fin_desde
   escribir('Existen', NE, 'letras b')
fin
```

Krasher Padres 1095564-2202172344
1095564220217234

8.9. *Escribir una función convertida (núm, b) que nos permita transformar un número entero y positivo en base 10 a la base que le indiquemos como parámetro. Comprobar el algoritmo para las bases 2 y 16.*

```
algoritmo Cambio_de_base
   var entero: núm, b
   inicio
      escribir('Déme número')
      leer(núm)
      escribir('Indique base')
      leer(b)
      escribir(convertir(núm, b),'es el número', núm, 'en base', b)
   fin

cadena función convertir(E entero: núm, b)
   var entero: r
      carácter: c
      cadena: unacadena
   inicio
      unacadena ← ''
      si núm > 0 entonces
         mientras núm > 0 hacer
            r ← núm MOD b
            si r > 9 entonces
               c ← car(r+55)
               si_no
               c ← car(r + código('0'))
            fin_si
            unacadena ← c + unacadena
            núm ← núm div b
         fin_mientras
      si_no
         unacadena ← '0'
   fin_si
   devolver(unacadena)
fin_función
```

Krasner Podres 1095564-2202172345

1095564220217234S

CONCEPTOS CLAVE

- Cadena
- Cadena nula
- Comparación de cadenas
- Concatenación
- Funciones de biblioteca
- Literal de cadena
- Longitud de la cadena
- *String*
- Variable de cadena

RESUMEN

Cada lenguaje de computadora tiene su propio método de manipulación de cadenas de caracteres. Algunos lenguajes, tales como C++ y C, tienen un conjunto muy rico de funciones de manipulación de cadenas. Otros lenguajes, tales como Fortran, que se utilizan predominantemente para cálculos numéricos, incorporan características de manipulación de cadenas en sus últimas versiones. También lenguajes como LISP, que está concebido para manipular aplicaciones de listas proporciona capacidades excepcionales de manipulación de cadenas.

En un lenguaje como C o C++, las cadenas son simplemente *arrays* de caracteres terminados en ca-

racteres nulos ("\0") que se pueden manipular utilizando técnicas estándares de procesamiento de *arrays* elemento por elemento. En esencia, las cadenas en los lenguajes de programación modernos tienen, fundamentalmente, estas características:

1. Una cadena (*string*) es un array de caracteres que en algunos casos (C++) se termina con el carácter `NULO (NULL)`.
2. Las cadenas se pueden procesar siempre utilizando técnicas estándares de procesamiento de arrays.
3. En la mayoría de los lenguajes de programación existen muchas funciones de biblioteca para procesamiento de cadenas como una unidad completa. Internamente estas funciones manipulan las cadenas carácter a carácter.
4. Algunos caracteres se escriben con un código de escape o secuencia de escape, que consta del carácter escape (\) seguido por un código del propio carácter.
5. Un carácter se representa utilizando un único byte (8 bits). Los códigos de caracteres estándar más utilizados en los lenguajes de programación son **ASCII** y **Unicode**.
6. El código ASCII representa 127 caracteres y el código ASCII ampliado representa 256 caracteres. Mediante el código Unicode se llegan a representar numerosos lenguajes internacionales, además del inglés, como el español, francés, chino, hindi, alemán, etcétera.
7. Las bibliotecas estándar de funciones incorporadas a los lenguajes de programación incluyen gran cantidad de funciones integradas que manipulan cadenas y que actúan de modo similar a los algoritmos de las funciones explicadas en el capítulo. Éste es el caso de la biblioteca de cadenas del lenguaje C o la biblioteca *string.h* de C++.
8. Algunas de las funciones de cadena típicas son: *longitud de la cadena, comparar cadenas, insertar cadena, copiar cadenas, concatenar cadenas*, etcétera.
9. El lenguaje C++ soporta las cadenas como arrays de caracteres terminado en el carácter nulo representado por la secuencia de escape "\0".
10. Los lenguajes orientados a objetos Java y C# soportan las cadenas como objetos de la clase `String`.

Krasner Podres 1095584-2202172346

EJERCICIOS

8.1. Escribir un algoritmo para determinar si una cadena especificada ocurre en una cadena dada, y si es así, escribir un asterisco (*) en la primera posición de cada ocurrencia.

8.2. Escribir un algoritmo para contar el número de ocurrencias de cada una de las palabras 'a', 'an' y 'and' en las diferentes líneas de texto.

8.3. Contar el número de ocurrencias de una cadena especificada en diferentes líneas de texto.

8.4. Escribir un algoritmo que permita la entrada de un nombre consistente en un nombre, un primer apellido y un segundo apellido, en ese orden, y que imprima a continuación el último apellido, seguido del primer apellido y el nombre. Por ejemplo: `Luis Garcia` producirá: `Garcia Garcia Luis`.

8.5. Escribir un algoritmo que elimine todos los espacios finales en una cadena determinada. Por ejemplo: 'J. R. GARCIA' se deberá transformar en 'J. R. GARCIA'.

8.6. Diseñar un algoritmo cuya entrada sea una cadena `S` y un factor de multiplicación `N`, cuya función sea generar la cadena dada `N` veces. Por ejemplo:

'¡Hey!', 3

se convertirá en:

'¡Hey! ¡Hey! ¡Hey!'

8.7. Diseñar un algoritmo que elimine todas las ocurrencias de cada carácter en una cadena dada a partir de otra cadena dada. Las dos cadenas son:

- `CADENA1` es la cadena donde deben eliminarse caracteres.
- `LISTA` es la cadena que proporciona los caracteres que deben eliminarse.

```
CADENA = 'EL EZNZZXTX'
LISTA = 'XZ'
```

la cadena pedida es 'EL ENT'.

8.8. Escribir un algoritmo que convierta los números arábigos en romanos y viceversa (I = 1, V = 5, X = 10, L = 50, C = 100, D = 500 y M = 1000).

8.9. Diseñar un algoritmo que mediante una función permita cambiar un número n en base 10 a la base b, siendo b un número entre 2 y 20.

8.10. Escribir el algoritmo de una función que convierta una cadena en mayúsculas y otra que la convierta en minúsculas.

8.11. Diseñar una función que informe si una cadena es un palíndromo (una cadena es un palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda).

Krasher Padres 1095564-2202172347

1095564220217234?

1095564-2202172347

CAPÍTULO 9

Archivos (ficheros)

9.1. Archivos y flujos (*stream*): la jerarquía de datos
9.2. Conceptos y definiciones: terminología
9.3. Soportes secuenciales y direccionables
9.4. Organización de archivos
9.5. Operaciones sobre archivos
9.6. Gestión de archivos
9.7. Flujos
9.8. Mantenimiento de archivos
9.9. Procesamiento de archivos secuenciales (algoritmos)
9.10. Procesamiento de archivos directos (algoritmos)
9.11. Procesamiento de archivos secuenciales indexados
9.12. Tipos de archivos: consideraciones prácticas en C/C++ y Java

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Krasner Padres 1095564-2202172358

INTRODUCCIÓN

Los datos que se han tratado hasta este capítulo y que son procesados por un programa pueden residir simultáneamente en la memoria principal de la computadora. Sin embargo, grandes cantidades de datos se almacenan normalmente en dispositivos de memoria auxiliar. Las diferentes técnicas que han sido diseñadas para la estructuración de estas colecciones de datos complejas se alojaban en arrays. En este capítulo se realiza una introducción a la organización y gestión de datos estructurados sobre dispositivos de almacenamiento secundario, tales como cintas y discos magnéticos. Estas colecciones de datos se conocen como **archivos (ficheros)**. Las técnicas requeridas para gestionar archivos son diferentes de las técnicas de organización de datos que son efectivas en la memoria principal, aunque se construyen sobre la base de esas técnicas. Este capítulo introductorio está concebido para la iniciación a los archivos, lo que son y sus misiones en los sistemas de información y de los problemas básicos en su organización y gestión.

9.1. Archivos y flujos (*stream*): la jerarquía de datos

El almacenamiento de datos en variables y ***arrays*** **(arreglos)** es temporal; los datos se pierden cuando una variable sale de su ámbito o alcance de influencia, o bien cuando se termina el programa. La mayoría de las aplicaciones requieren que la información se almacene de forma persistente, es decir, que no se borre o elimine cuando se termina la ejecución del programa. Por otra parte, en numerosas aplicaciones es necesario utilizar grandes cantidades de información que, normalmente, no caben en la memoria principal. Debido a estas causas se requiere utilizar **archivos (ficheros)** para almacenar de modo permanente grandes cantidades de datos, incluso después que los programas que crean los datos se terminan. Estos datos almacenados en archivos se conocen como **datos persistentes** y permanecen después de la duración de la ejecución del programa.

Las computadoras almacenan los archivos en dispositivos de almacenamiento secundarios, como discos CD, DVD, memorias *flash* USB, memorias de cámaras digitales, etcétera o en la nube.. En este capítulo se explicará cómo los programas escritos en un lenguaje de programación crean, actualizan o procesan archivos de datos.

El procesamiento de archivos es una de las características más importantes que un lenguaje de programación debe tener para soportar aplicaciones comerciales que procesan, normalmente, cantidades masivas de datos persistentes. La entrada de datos normalmente se realiza a través del teclado y la salida o resultados van a la pantalla. Estas operaciones, conocidas como Entrada/Salida (E/S), se realizan también hacia y desde los archivos.

Los programas que se crean con C/C++, Java u otros lenguajes necesitan interactuar con diferentes fuentes de datos. Los lenguajes antiguos como Fortran, Pascal o Cobol tenían integradas en el propio lenguaje las entradas y salidas; palabras reservadas como `PRINT`, `READ`, `write`, `writeln`, etcétera, son parte del vocabulario del lenguaje. Sin embargo, los lenguajes de programación modernos como C/C++ o Java/C# tienen entradas y salidas en el lenguaje y para acceder o almacenar información en una unidad de disco duro o en un CD o en un DVD, en páginas de un *sitio web* e incluso guardar bytes en la memoria de la computadora, se necesitan técnicas que pueden ser diferentes para cada dispositivo de almacenamiento. Afortunadamente, los lenguajes citados anteriormente pueden almacenar y recuperar información, utilizando sistemas de comunicaciones denominados **flujos** que se implementan en bibliotecas estándar de funciones de E/S (en archivos de cabecera `stdio.h` y `cstdio.h`) en C, en una biblioteca estándar de clases (en archivos de cabecera `iostream` y `fstream`) en C++, o en el paquete Java.io en el lenguaje Java.

Las estructuras de datos enunciadas en los capítulos anteriores se encuentran almacenadas en la memoria central o principal. Este tipo de almacenamiento, conocido como *almacenamiento principal* o *primario*, tiene la ventaja de su pequeño tiempo de acceso y, además, que este tiempo necesario para acceder a los datos almacenados en una posición es el mismo que el tiempo necesario para acceder a los datos almacenados en otra posición del dispositivo –memoria principal–. Sin embargo, no siempre es posible almacenar los datos en la memoria central o principal de la computadora, debido a las limitaciones que su uso plantea:

- La cantidad de datos que puede manipular un programa no puede ser muy grande debido a la limitación de la memoria central de la computadora.[1]
- La existencia de los datos en la memoria principal está supeditada al tiempo que la computadora está encendida y el programa ejecutándose (tiempo de vida efímero). Esto supone que los datos desaparecen de la memoria principal cuando la computadora se apaga o se deja de ejecutar el programa.

Estas limitaciones dificultan:

- La manipulación de gran número de datos, ya que –en ocasiones– puede no caber en la memoria principal (aunque hoy en día han desaparecido las limitaciones que la primera generación de PC presentaba: una memoria de 640 KB. Las memorias RAM actuales oscilan entre 4 y 32 GB o más en computadoras de escritorio y laptops (portátiles) y 32 a 128 GB en estaciones de trabajo.
- La transmisión de salida de resultados de un programa pueda ser tratada como entrada a otro programa.

Para poder superar estas dificultades se necesitan dispositivos de almacenamiento secundario (memorias externas o auxiliares) como discos duros, CD/DVD o la nube, entre otros, donde se almacenará la información o datos que

[1] En sus orígenes y en la década de los ochenta, 640 K-bytes en el caso de las computadoras personales IBM PC y compatibles. Hoy día esas cifras han sido superadas con creces, pero aunque las memorias centrales varían, en computadoras domésticas, portátiles (*laptops*) y de escritorio, entre 4 GB y 32 GB, la temporalidad de los datos almacenados en ellas aconseja siempre el uso de archivos para datos de carácter permanente.

Krasher Padres 1995564-2202172359

podrán ser recuperados para su tratamiento posterior. Las estructuras de datos aplicadas a colección de datos en almacenamientos secundarios se llaman *organización de archivos*. La noción de ***archivo*** o ***fichero*** está relacionada con los conceptos de:

- Almacenamiento permanente de datos.
- Fraccionamiento o partición de grandes volúmenes de información en unidades más pequeñas que puedan ser almacenadas en memoria central y procesadas por un programa.

Un *archivo* o *fichero* es un conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas *registros* o *artículos*, que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajo denominadas *campos*.

9.1.1. Campos

Un *campo* es un *ítem* o *elemento de datos elementales*, tales como un nombre, número de empleados, ciudad, número de identificación, entre otros.

Un campo está caracterizado por su tamaño o longitud y su tipo de datos (cadena de caracteres, entero, lógico, etcétera). Los campos pueden incluso variar en longitud. En la mayoría de los lenguajes de programación los campos de longitud variable no están soportados y se suponen de longitud fija.

Campos

Nombre	Dirección	Fecha de nacimiento	Estudios	Salario	Trienios

Figura 9.1. Campos de un registro.

Un campo es la unidad mínima de información de un registro; los datos contenidos en él se dividen con frecuencia en *subcampos*; por ejemplo, el campo fecha se divide en los subcampos día, mes, año.

Campo	0	7	0	7	1	9	9	5
Subcampo	Día		Mes		Año			

Los rangos numéricos de variación de los subcampos anteriores son:

```
1 ≤ día ≤ 31
1 ≤ mes ≤ 12
1 ≤ año ≤ 1987
```

9.1.2. Registros

Un *registro* es una colección de información, normalmente relativa a una entidad particular; es una colección de campos lógicamente relacionados que pueden ser tratados como una unidad por algún programa. Un ejemplo de un registro puede ser la información de un determinado empleado que contiene los campos de nombre, dirección, fecha de nacimiento, estudios, salario, trienios, entre otros.

Los registros pueden ser de *longitud fija*; por ejemplo, los registros de empleados pueden contener el mismo número de campos, cada uno de la misma longitud para nombre, dirección, fecha, etcétera. También pueden ser de *longitud variable*.

Los registros organizados en campos se denominan *registros lógicos*.

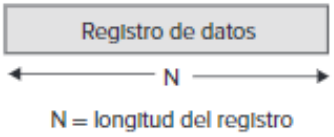


Figura 9.2. Registro.

Krasner Padres 1095564-2202172359

Nota

El concepto de registro es similar al de estructura (`struct`) estudiado en el capítulo 7, ya que ambos permiten almacenar datos de tipo heterogéneo.

9.1.3. Archivos (ficheros)

Un *fichero* (*archivo*) de datos –o simplemente un **archivo**– es una colección de registros relacionados entre sí con aspectos en común y organizados para un propósito específico; por ejemplo, un fichero de una clase escolar contiene un conjunto de registros de los estudiantes de esa clase. Otros ejemplos pueden ser el fichero de nóminas de una empresa, inventarios, *stocks*, etcétera.

La figura 9.3 recoge la estructura de un archivo correspondiente a los suscriptores de una revista de informática.

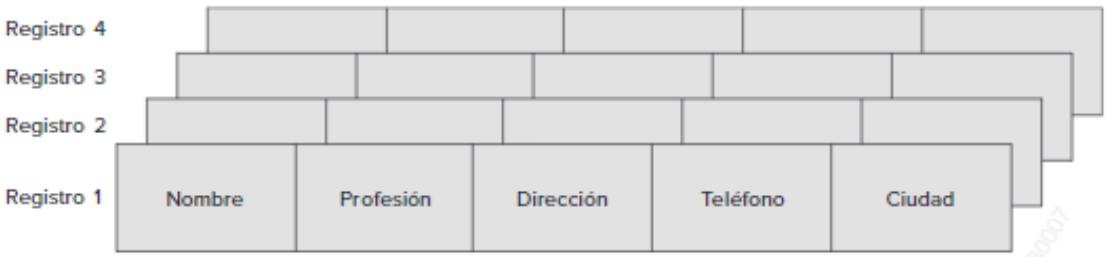


Figura 9.3. Estructuras de un archivo "suscriptores".

Un archivo en una computadora es una estructura diseñada para contener datos que están organizados de tal modo que puedan ser recuperados fácilmente, actualizados o borrados y almacenados de nuevo en el archivo con todos los campos realizados.

9.1.4. Bases de datos

Es una colección de archivos a los que puede accederse por un conjunto de programas y que contienen todos ellos datos relacionados; por ejemplo, una base de datos de una universidad puede contener archivos de estudiantes, archivos de nóminas, inventarios de equipos, etcétera.

9.1.5. Estructura jerárquica

Carácter, campos, registro, archivo y base de datos son *conceptos lógicos* que se refieren al medio en que el usuario de computadoras ve los datos y se organizan. Las estructuras de datos se organizan de un modo jerárquico, de manera que el nivel más alto lo constituye la base de datos y el nivel más bajo el carácter.

9.1.6. Jerarquía de datos

Una computadora, como ya conoce el lector (capítulo 1), procesa todos los datos como combinaciones de ceros y unos; tal elemento de los datos se denomina bit (*binary digit*). Sin embargo, como se puede deducir fácilmente, es difícil para los programadores trabajar con datos en estos formatos de bits de bajo nivel; en su lugar, los programadores prefieren trabajar con caracteres como los dígitos decimales (`0-9`), letras (`A-Z` y `a-z`) o símbolos especiales (`&, *, , @, €, #,...`). El conjunto de todos los caracteres utilizados para escribir los programas se denomina *conjunto* o *juegos de caracteres* de la computadora; cada carácter se representa como un patrón de ceros y unos, por ejemplo, en Java, los caracteres son Unicode (capítulo 1) compuestos de 2 bytes.

Krasner Podres 1095584-2202180007

Así como los caracteres se componen de bits, los *campos* se componen de caracteres o bytes. Un **campo** es un grupo de caracteres o bytes que representan un significado; por ejemplo, un campo puede constar de letras mayúsculas y minúsculas que representan el nombre de una ciudad.

Los datos procesados por las computadoras se organizan en *jerarquías de datos* formando estructuras a partir de bits, caracteres, campos, etcétera.

Los campos (variables de instancias en C++ y Java) se agrupan en *registros* que se implementan en una **clase** en Java o en C++. Un registro es un grupo de campos relacionados que se implementan con tipos de datos básicos o estructurados. En un sistema de matrícula en una universidad, un registro de un alumno o de un profesor puede constar de los siguientes campos:

- Nombre `(cadena)`.
- Número de expediente `(entero)`.
- Número de documento nacional de identidad o pasaporte `(entero doble)`.
- Año de nacimiento `(entero)`.
- Estudios `(cadena)`.

Un archivo es un grupo de registros relacionados; por ejemplo, una universidad puede tener muchos alumnos y profesores, y un archivo de alumnos contiene un registro para cada empleado. Un archivo de una universidad puede contener miles de registros y millones o incluso miles de millones de caracteres de información. Las figura 9.4 muestra la *jerarquía de datos* de un archivo (*byte, campo, registro, archivo*).

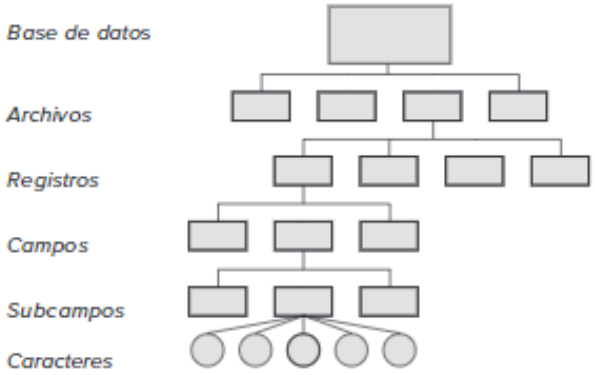


Figura 9.4. Estructuras jerárquicas de datos.

Los registros poseen una *clave* o llave que identifica a cada uno y que es única para diferenciarla de otros. En registros de nombres es usual que el campo clave sea el pasaporte o el DNI (Documento Nacional de Identidad).

Un conjunto de archivos relacionados se denomina base de datos; en los negocios o en la administración, los datos se almacenan en bases de datos y en muchos archivos diferentes. Por ejemplo, las universidades pueden tener archivos de profesores, de estudiantes, de planes de estudio, de nóminas de profesores y de PAS (Personal de Administración y Servicios). Otra jerarquía de datos son los sistemas de gestión de bases de datos (**SGBD** o **DBMS**) que es un conjunto de programas diseñados para crear y administrar bases de datos.

9.2. Conceptos y definiciones: terminología

Aunque en el apartado anterior ya se han comentado algunos términos relativos a la teoría de archivos, en este apartado se enunciarán los términos más utilizados en la gestión y diseño de archivos.

9.2.1. Clave (indicativo)

Una *clave* (*key*) o *indicativo* es un campo de datos que identifica el registro y lo diferencia de otros registros; esta clave debe ser diferente para cada registro, las claves típicas son nombres o números de identificación.

Krasner Padres 1095564-2202180008

9.2.2. Registro físico o bloque

Un *registro físico* o *bloque* es la cantidad más pequeña de datos que pueden transferirse en una operación de entrada/salida entre la memoria central y los dispositivos periféricos o viceversa, ejemplos de registros físicos son: una tarjeta perforada, una línea de impresión, un sector de un disco magnético, etcétera.

Un bloque puede contener uno o más registros lógicos, éstos pueden ocupar menos de un registro físico, un registro físico o más de un registro físico.

9.2.3. Factor de bloqueo

Otra característica que es importante en relación con los archivos es el concepto de *factor de bloqueo* o *blocaje*. El número de registros lógicos que puede contener un registro físico se denomina factor de bloqueo.

Se pueden dar las siguientes situaciones:

- *Registro lógico* > *Registro físico*. En un bloque se contienen varios registros físicos; se denominan registros expandidos.
- *Registro lógico* = *Registro físico*. El factor de bloqueo es 1 y se dice que los registros *no están bloqueados*.
- *Registro lógico* < *Registro físico*. El factor de bloqueo es mayor que 1 y los registros *están bloqueados*.

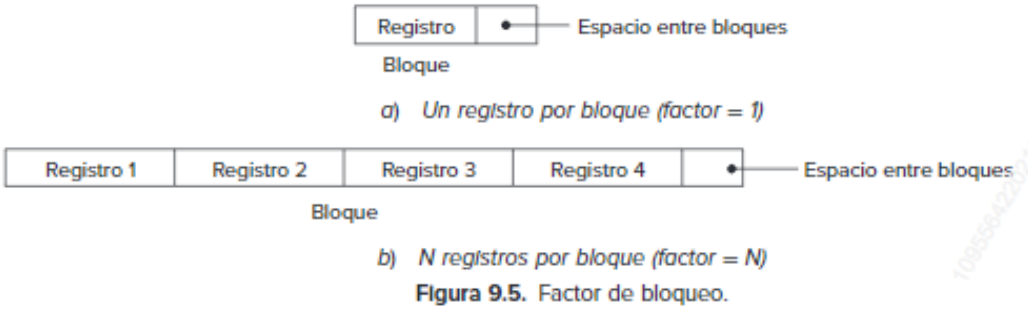


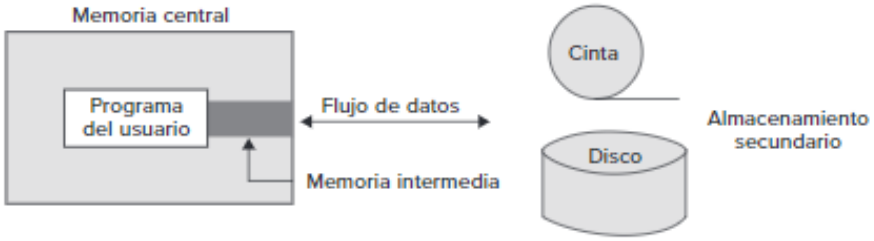
Figura 9.5. Factor de bloqueo.

La importancia del factor de bloqueo se puede apreciar mejor con un ejemplo: supongamos que se tienen dos archivos, uno de ellos tiene un factor de bloqueo de 1 (un registro en cada bloque), el otro archivo tiene un factor de bloqueo de 10 (10 registros/bloque). Si cada archivo contiene un millón de registros, el segundo archivo requerirá 900000 operaciones de entrada/salida menos para leer todos los registros. En el caso de las computadoras personales con un tiempo medio de acceso de 90 milisegundos, el primer archivo emplearía alrededor de 24 horas más para leer todos los registros del archivo.

Un factor de bloqueo mayor que 1 siempre mejora el rendimiento; entonces, ¿por qué no incluir todos los registros en un solo bloque? La razón reside en que las operaciones de entrada/salida que se realizan por bloques se hacen a través de un área de la memoria central denominada *memoria intermedia (buffer)* y entonces el aumento del bloque implicará aumento de la memoria intermedia y, por consiguiente, se reducirá el tamaño de la memoria central.

El tamaño de una memoria intermedia de un archivo es el mismo que el del tamaño de un bloque. Como la memoria central es más cara que la secundaria, no conviene aumentar el tamaño del bloque alegremente, sino más bien conseguir un equilibrio entre ambos criterios.

En el caso de las computadoras personales, el registro físico puede ser un sector del disco (512 bytes).



Krasner Podres 1095584-2202160009

La tabla 9.1 resume los conceptos lógicos y físicos de un registro.

Tabla 9.1. Unidades de datos lógicos y físicos.

Organización lógica	Organización física	Descripción
	Bit	Dígito binario.
Carácter	Byte (octeto, 8 bits)	En la mayoría de los códigos un carácter se representa aproximadamente por un byte.
Campo	Palabra	Es un conjunto relacionado de caracteres; una palabra de computadora es un número fijo de bytes.
Registro	Bloque (1 página = bloques de longitud)	Los registros pueden estar bloqueados.
Archivo	Área	Varios archivos se pueden almacenar en un área de almacenamiento.
Base de datos	Áreas	Colección de archivos de datos relacionados que se pueden organizar en una base de datos.

Resumen de archivos

- Un archivo está siempre almacenado en un soporte externo a la memoria central.
- Existe independencia de las informaciones respecto de los programas.
- Todo programa de tratamiento intercambia información con el archivo y la unidad básica de entrada/salida es el registro.
- La información almacenada es permanente.
- En un momento dado, los datos extraídos por el archivo son los de un registro y no los del archivo completo.
- Los archivos en memoria auxiliar permiten una gran capacidad de almacenamiento.

Krasnar Padres 1095564-2202180010

9.3. Soportes secuenciales y direccionables

El soporte es el medio físico donde se almacenan los datos, los tipos de soporte utilizados en la gestión de archivos son:

- *Soportes secuenciales.*
- *Soportes direccionables.*

Los *soportes secuenciales* son aquellos en los que los registros –informaciones– están escritos unos a continuación de otros y para acceder a un determinado registro n se necesita pasar por los $n - 1$ registros anteriores.

Los *soportes direccionables* se estructuran de modo que la información registrada se puede localizar directamente por su dirección y no se requiere pasar por los registros precedentes. En estos soportes los registros deben poseer un campo clave que los diferencie del resto de los registros del archivo. Una dirección en un soporte direccionable puede ser número de pista y número de sector en un disco.

Los soportes direccionables son los discos magnéticos, aunque pueden actuar como soporte secuencial.

9.4. Organización de archivos

Según las características del soporte empleado y el modo como se han organizado los registros, se consideran dos tipos de acceso a los registros de un archivo:

- *Acceso secuencial.*
- *Acceso directo.*

El *acceso secuencial* implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro.

El *acceso directo* implica el acceso a un registro determinado, sin que ello implique la consulta de los registros precedentes. Este tipo de acceso sólo es posible con soportes direccionables.

La *organización* de un archivo define la forma en la que los registros se disponen sobre el soporte de almacenamiento, o también se define la organización como la forma en que se estructuran los datos en un archivo. En general, se consideran tres organizaciones fundamentales:

- *Organización secuencial.*
- *Organización directa o aleatoria (random).*
- *Organización secuencial indexada (indexed).*

9.4.1. Organización secuencial

Un archivo con organización secuencial es una sucesión de registros almacenados consecutivamente sobre el soporte externo, de tal modo que para acceder a un registro n dado es obligatorio pasar por todos los $n - 1$ registros que le preceden.

Los registros se graban consecutivamente cuando el archivo se crea y se debe acceder consecutivamente cuando se leen dichos registros.

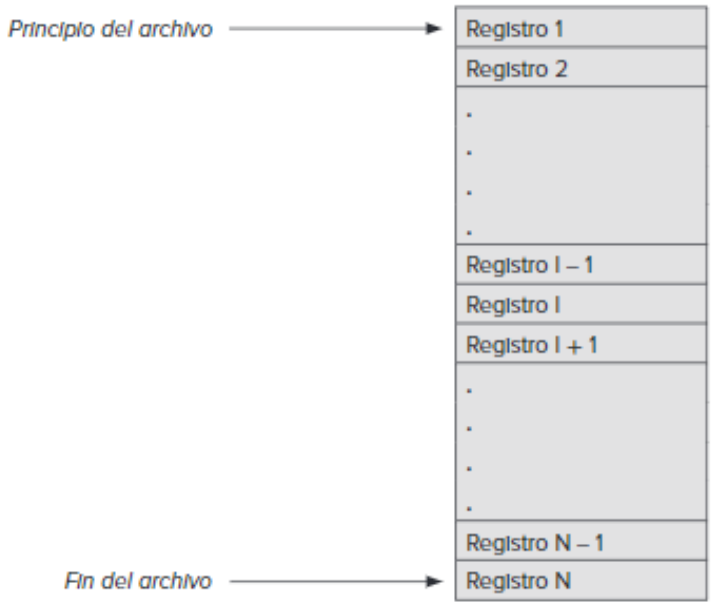


Figura 9.6. Organización secuencial.

- El orden físico en que fueron grabados (escritos) los registros es el orden de lectura de los mismos.
- Todos los tipos de dispositivos de memoria auxiliar soportan la organización secuencial.

Los archivos organizados secuencialmente contienen un registro particular —el último— que contiene una marca fin de archivo (`EOF` o bien `FF`). Esta marca fin de archivo puede ser un carácter especial como '*'.

9.4.2. Organización directa

Un archivo está organizado en modo directo cuando el orden físico no se corresponde con el orden lógico. Los datos se sitúan en el archivo y se accede a ellos directamente mediante su posición, es decir, el lugar relativo que

Krasher Podres 1095584-2202160011

ocupan. Esta organización tiene la *ventaja* de que se pueden leer y escribir registros en cualquier orden y posición; es muy rápido acceder a la información que contienen.

La organización directa tiene el *inconveniente* de que necesita programar la relación existente entre el contenido de un registro y la posición que ocupa. El acceso a los registros en modo directo implica la posible existencia de huecos libres dentro del soporte y, por consecuencia, pueden existir huecos libres entre registros.

La correspondencia entre clave y dirección debe poder ser programada y la determinación de la relación entre el registro y su posición física se obtiene mediante una fórmula.

Las condiciones para que un archivo sea de organización directa son:

- Almacenado en un soporte direccionable.
- Los registros deben contener un campo específico denominado *clave* que identifica cada registro de modo único, es decir, dos registros distintos no pueden tener un mismo valor de clave.
- Existencia de una correspondencia entre los posibles valores de la clave y las direcciones disponibles sobre el soporte.

Un soporte direccionable es normalmente un disco o paquete de discos. Cada posición se localiza por su *dirección absoluta*, que en el caso del disco suele venir definida por dos parámetros –número de pista y número de sector– o bien por tres parámetros –pista, sector y número de cilindro–; un *cilindro i* es el conjunto de pistas de número *i* de cada superficie de almacenamiento de la pila.

En la práctica el programador no gestiona directamente direcciones absolutas, sino *direcciones relativas* respecto al principio del archivo. La manipulación de direcciones relativas permite diseñar el programa con independencia de la posición absoluta del archivo en el soporte.

El programador crea una relación perfectamente definida entre la clave indicativa de cada registro y su posición física dentro del dispositivo de almacenamiento; en ocasiones, esta relación produce *colisiones*.

A continuación, consideremos el fenómeno de las *colisiones* mediante un ejemplo.

La clave de los registros de estudiantes de una Facultad de Ciencias es el número de expediente escolar que se le asigna en el momento que se matricula y que consta de ocho dígitos. Si el número de estudiantes es un decimal de ocho dígitos, existen 10^8 posibles estudiantes (0 a 99999999), aunque lógicamente *nunca* existirán tantos (incluso incluyendo alumnos ya graduados). El archivo de estudiantes constará a lo sumo de decenas o centenas de miles de estudiantes. Se desea almacenar este archivo en un disco sin utilizar mucho espacio; en caso de querer obtener el algoritmo de direccionamiento, será necesaria una *función de conversión de claves o función hash*. Suponiendo que N es el número de posiciones disponibles para el archivo, el algoritmo de direccionamiento convierte cada valor de la clave en una dirección relativa *d*, comprendida entre 1 y *N*. Como la clave puede ser numérica o alfanumérica, el algoritmo de conversión debe prever esta posibilidad y asignar a cada registro correspondiente a una clave una posición física en el soporte de almacenamiento. Asimismo, el algoritmo o función de conversión de claves debe eliminar o reducir al máximo las colisiones. Se dice que en un algoritmo de conversión de claves, se produce una *colisión* cuando dos registros de claves distintas producen la misma dirección física en el soporte. El *inconveniente de una colisión* radica en el hecho de tener que situar el registro en una posición diferente de la indicada por el algoritmo de conversión y, por consiguiente, el acceso a este registro será más lento. Las colisiones son difíciles de evitar en las organizaciones directas. Sin embargo, un tratamiento adecuado en las operaciones de lectura/escritura disminuirá su efecto perjudicial en el archivo.

Para representar la función de transformación o conversión de claves (*hash*), se puede utilizar una notación matemática. Así, si K es una clave, f(K) es la correspondiente dirección; f es la función llamada *función de conversión*.

EJEMPLO 9.1

Una compañía de empleados tiene un número determinado de vendedores y un archivo en el que cada registro corresponde a un vendedor. Existen 200 vendedores, cada uno referenciado por un número de cinco dígitos. Si tuviésemos que asignar un archivo de 100000 registros, cada uno se corresponderá con una posición del disco.

Para el diseño del archivo crearemos 250 registros (25% más que el número de registros necesarios –25 por 100 suele ser un porcentaje habitual–) que se distribuirán de la siguiente forma:

1. Posiciones 0-199 constituyen el área principal del archivo y en ella se almacenarán todos los vendedores.

Krasher Padres 1095564-2202180013

2. Posiciones 200-249 constituyen el área de desbordamiento, si $K(1) <> K(2)$, pero $f(K(1)) = f(K(2))$, y el registro con clave $K(1)$ ya está almacenado en el área principal, entonces el registro con $K(2)$ se almacena en el área de desbordamiento.

La función f se puede definir como:

$f(k)$ = resto cuando K se divide por 199, esto es, el módulo de 199; 199 ha sido elegido por ser el número primo mayor y que es menor que el tamaño del área principal.

Para establecer el archivo se borran primero 250 posiciones. A continuación, para cada registro de vendedor se calcula $p = f(K)$. Si la posición p está vacía, se almacena el registro en ella, en caso contrario se busca secuencialmente a través de las posiciones 200, 201, ..., para el registro con la clave deseada.

9.4.3. Organización secuencial indexada

Un diccionario es un archivo secuencial, cuyos registros son las entradas y sus claves son las palabras definidas por las entradas. Para buscar una palabra (una clave) no se busca secuencialmente desde la "a" hasta la "z", sino que se abre el diccionario por la letra inicial de la palabra. Si se desea buscar "índice", se abre el índice por la letra *I* y en su primera página se busca la cabecera de página hasta encontrar la más próxima a la palabra, buscando a continuación palabra a palabra hasta encontrar "índice". El diccionario es un ejemplo típico de archivo secuencial indexado con dos niveles de índices, el nivel superior para las letras iniciales y el nivel menor para las cabeceras de página. En una organización de computadora las letras y las cabeceras de páginas se guardarán en un archivo de índice independiente de las entradas del diccionario (archivo de datos). Por consiguiente, cada archivo secuencial indexado consta de un archivo índice y un archivo de datos.

Un archivo está organizado en forma secuencial indexada si:

- El tipo de sus registros contiene un campo clave identificador.
- Los registros están situados en un soporte direccionable por el orden de los valores indicados por la clave.
- Un índice para cada posición direccionable, la dirección de la posición y el valor de la clave; en esencia, el índice contiene la clave del último registro y la dirección de acceso al primer registro del bloque.

Un archivo en organización secuencial indexada consta de las siguientes partes:

- Área *de datos o primaria*: contiene los registros en forma secuencial y está organizada en secuencia de claves sin dejar huecos intercalados.
- Área *de índices*: es una tabla que contiene los niveles de índice, la existencia de varios índices enlazados se denomina *nivel de indexación*.
- Área *de desbordamiento o excedentes*: utilizada, si fuese necesario, para las actualizaciones.

El área de índices es equivalente, en su función, al índice de un libro. En ella se refleja el valor de la clave identificativa más alta de cada grupo de registros del archivo y la dirección de almacenamiento del grupo.

Los archivos secuenciales indexados presentan las siguientes *ventajas*:

- Rápido acceso.
- El sistema de gestión de archivos se encarga de relacionar la posición de cada registro con su contenido mediante la tabla de índices.

Y los siguientes *inconvenientes*:

- Desaprovechamiento del espacio por quedar huecos intermedios cada vez que se actualiza el archivo.
- Se necesita espacio adicional para el área de índices.

Los soportes que se utilizan para esta organización son los que permiten el acceso directo —los discos magnéticos—. Los soportes de acceso secuencial no pueden utilizarse, ya que no disponen de direcciones para las posiciones de almacenamiento.

Krasner Podres 1095564-2202180014

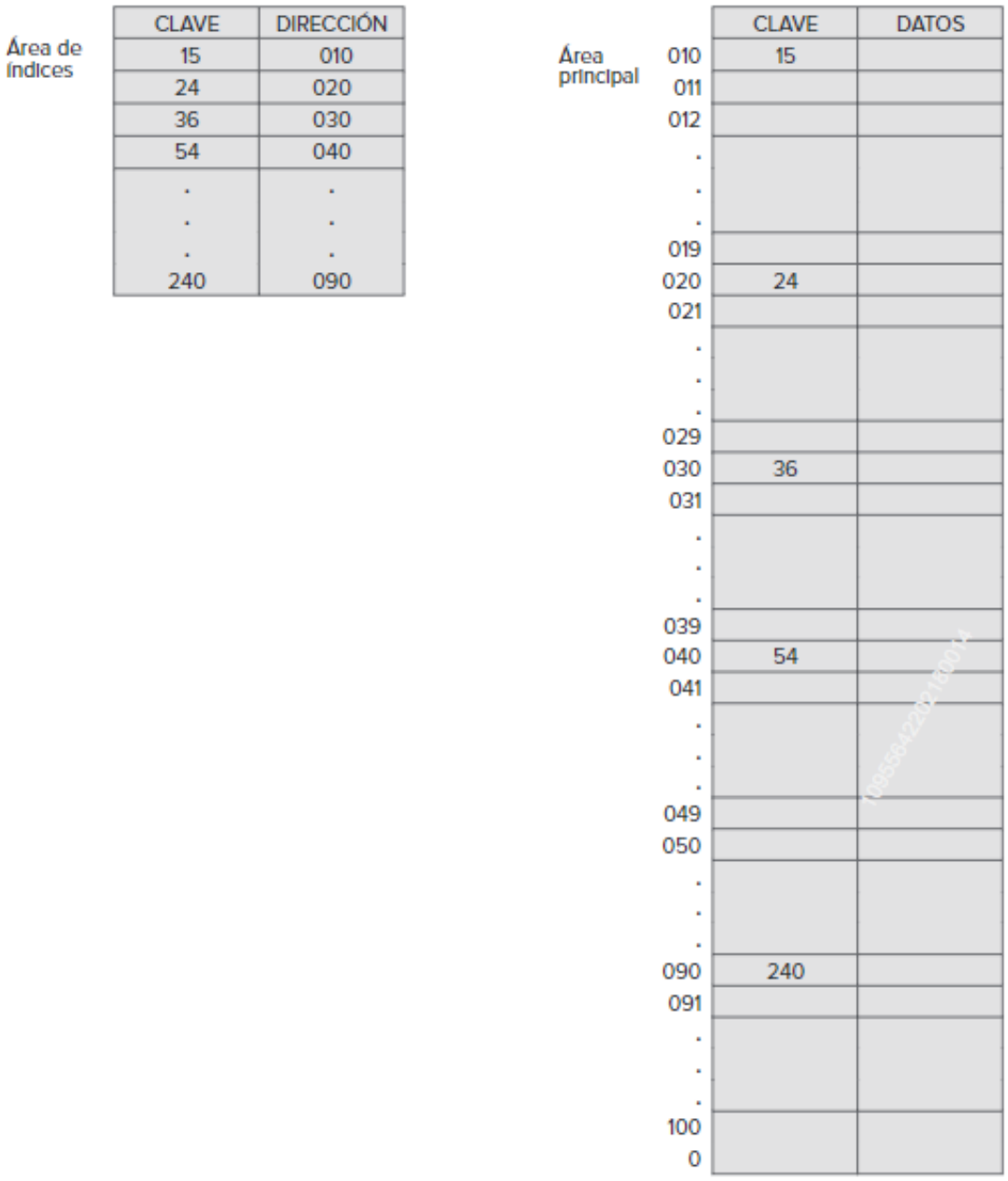


Figura 9.7. Organización secuencial indexada.

9.5. Operaciones sobre archivos

Tras la decisión del tipo de organización que ha de tener el archivo y los métodos de acceso que se van a aplicar para su manipulación, es preciso considerar todas las posibles operaciones que conciernen a los registros de un archivo. Las distintas operaciones que se pueden realizar son:

- *Creación.*
- *Consulta.*

Krasner Podres 1095564-2202180014

- *Actualización* (altas, bajas, modificación, consulta).
- *Clasificación*.
- *Reorganización*.
- *Destrucción* (borrado).
- *Reunión, fusión*.
- *Rotura, estallido*.

9.5.1. Creación de un archivo

Es la primera operación que sufrirá el archivo de datos. Implica la elección de un entorno descriptivo que permita un ágil, rápido y eficaz tratamiento del archivo.

Para utilizar un archivo, éste tiene que existir, es decir, la información de este archivo tiene que haber sido almacenada sobre un soporte y ser utilizable. La *creación* exige organización, estructura, localización o reserva de espacio en el soporte de almacenamiento, transferencia del archivo del soporte antiguo al nuevo.

Un archivo puede ser creado por primera vez en un soporte, proceder de otro previamente existente en el mismo o diferente soporte, ser el resultado de un cálculo o ambas cosas a la vez.

La figura 9.8 muestra un organigrama de la creación de un archivo de empleados de una empresa ordenado por el campo clave (número o código de empleado).

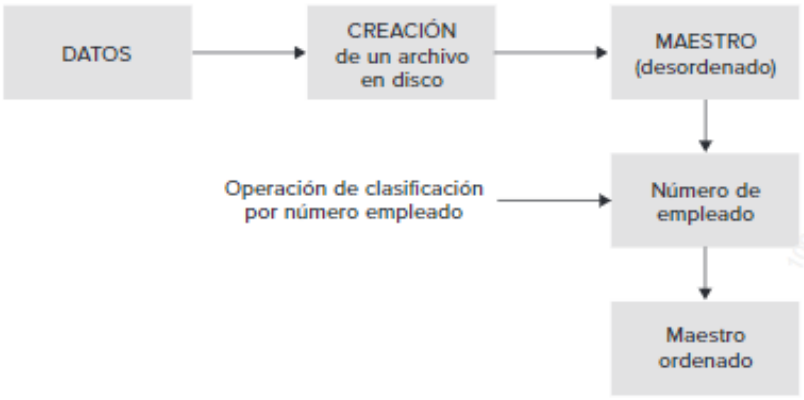


Figura 9.8. Creación de un archivo ordenado de empleados.

9.5.2. Consulta de un archivo

Es la operación que permite al usuario acceder al archivo de datos para conocer el contenido de uno, varios o todos los registros.

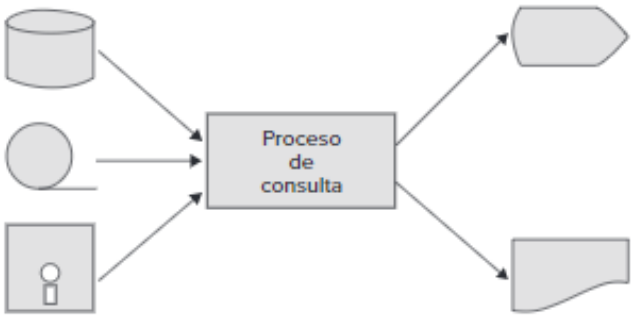


Figura 9.9. Consulta de un archivo.

Krasher Podres 1095564-2202180015

9.5.3. Actualización de un archivo

Es la operación que permite tener actualizado (puesto al día) el archivo, de tal modo que sea posible realizar las siguientes operaciones con sus registros:

- *Consulta* del contenido de un registro.
- *Inserción* de un registro nuevo en el archivo.
- *Supresión* de un registro existente.
- *Modificación* de un registro.

Un ejemplo de actualización es el de un archivo de un almacén, cuyos registros contienen las existencias de cada artículo, precios, proveedores, etcétera. Las existencias, precios, y demás, varían continuamente y exigen una actualización simultánea del archivo con cada operación de consulta.

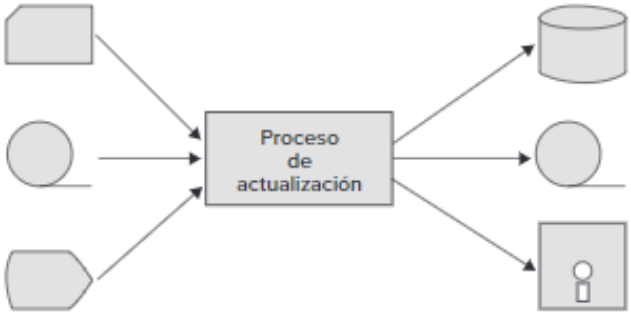


Figura 9.10. Actualización de un archivo (I).

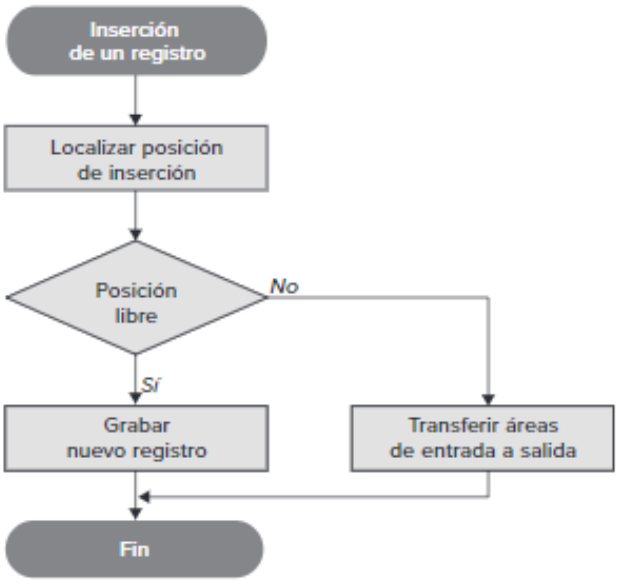


Figura 9.11. Actualización de un archivo (II).

9.5.4. Clasificación de un archivo

Una operación muy importante en un archivo es la *clasificación* u *ordenación* (*sort*, en inglés). Esta clasificación se realizará de acuerdo con el valor de un campo específico, pudiendo ser *ascendente* (creciente) o *descendente* (decreciente): alfabética o numérica (figura 9.12).

Krasner Podres 1095584-2202180016

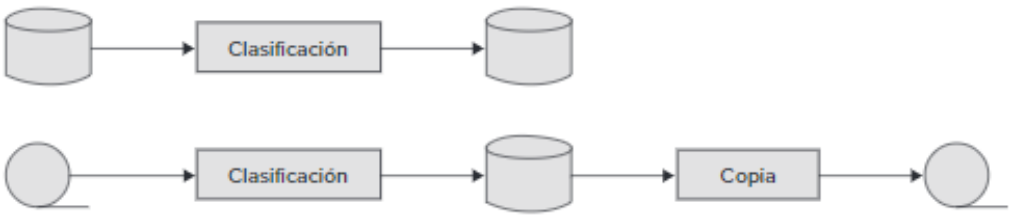


Figura 9.12. Clasificación de un archivo.

9.5.5. Reorganización de un archivo

Las operaciones sobre archivos modifican la estructura inicial o la óptima de un archivo. Los índices, enlaces (punteros o apuntadores), zonas de sinónimos, zonas de desbordamiento, etcétera, se modifican con el paso del tiempo, lo que hace a la operación de acceso al registro cada vez más lenta.

La reorganización suele consistir en la copia de un nuevo archivo a partir del archivo modificado, a fin de obtener una nueva estructura lo más óptima posible.

Kristher Podnes 1095584-2202180125

9.5.6. Destrucción de un archivo

Es la operación inversa a la creación de un archivo (*kill*, en inglés); cuando se destruye (anula o borra) un archivo, éste ya no se puede utilizar y, por consiguiente, no se podrá acceder a ninguno de sus registros (figura 9.13).

9.5.7. Reunión, fusión de un archivo

Reunión. Esta operación permite obtener un archivo a partir de otros (figura 9.14).

Fusión. Se realiza cuando se reúnen varios archivos en uno solo, intercalándose unos en otros, siguiendo criterios determinados.

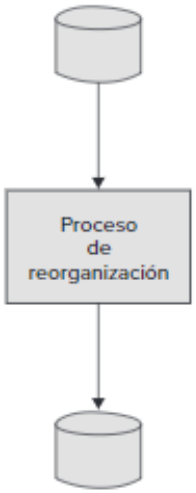


Figura 9.13. Reorganización de un archivo.

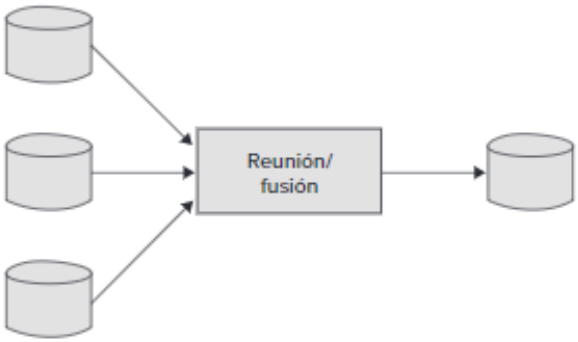


Figura 9.14. Fusión de archivos.

9.5.8. Rotura/estallido de un archivo

Es la operación de obtener varios archivos a partir de un mismo archivo inicial.

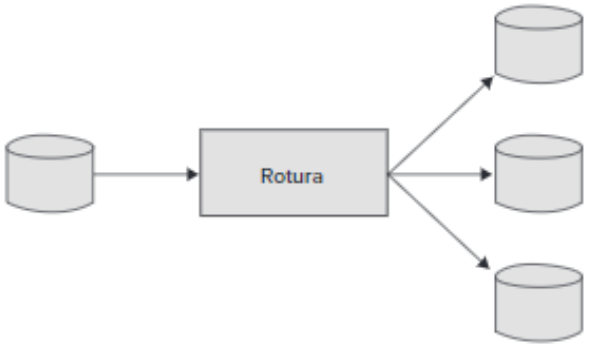


Figura 9.15. Rotura de un archivo.

Krasner Podres 1095584-2202180125

9.6. Gestión de archivos

Las operaciones sobre archivos se realizan por medio de programas y el primer paso para poder gestionarlos mediante un programa es declarar un identificador lógico que se asocie con el nombre externo del archivo para permitir su manipulación. La declaración se realizará con una serie de instrucciones como las que se muestran a continuación, cuya asociación permite establecer la organización del archivo y estructura de sus registros lógicos.

```
tipo
   registro: <tipo_registro>
      <tipo>:<nombre del campo>
       ....
   fin_registro
archivo_<organización> de <tipo_de_dato>:<tipo_archivo>

var
   tipo_registro: nombre_registro
   tipo_archivo:identificador_archivo

tipo
   registro: Rempleado
    cadena : nombre
    cadena : cod
    entero : edad
    real   : salario
   fin_registro
archivo_d de rempleado:empleado
var
   Rempleado: Re
   Empleado: E
```

Las operaciones básicas para la gestión de archivos que *tratan con la propia estructura del archivo* se consideran predefinidas y son:

- *Crear archivos (create).* Consiste en definirlo mediante un nombre y unos atributos; si el archivo existiera con anterioridad lo destruiría.

- *Abrir o arrancar (open)* un archivo que fue creado con anterioridad a la ejecución de este programa. Esta operación establece la comunicación de la CPU con el soporte físico del archivo, de forma que los registros se vuelven accesibles para lectura, escritura o lectura/escritura.
- *Incrementar* o *ampliar* el tamaño del archivo (*append, extend*).
- *Cerrar* el archivo después que el programa ha terminado de utilizarlo (*close*). Cierra la comunicación entre la CPU y el soporte físico del archivo.
- *Borrar (delete)* un archivo que ya existe. Borra el archivo del soporte físico, liberando espacio.
- *Transferir datos desde (leer) o a (escribir)* el dispositivo diseñado por el programa. Estas operaciones copian los registros del archivo sobre variables en memoria central y viceversa.

En cuanto a las operaciones más usuales en los registros son:

- *Consulta*: lectura del contenido de un registro.
- *Modificación*: alterar la información contenida en un registro.
- *Inserción*: añadir un nuevo registro al archivo.
- *Borrado*: suprimir un registro del archivo.

9.6.1. Crear un archivo

La creación de un archivo es la operación mediante la cual se introduce la información correspondiente al archivo en un soporte de almacenamiento de datos.

Antes de que cualquier usuario pueda procesar un archivo es preciso que éste haya sido creado previamente. El proceso de creación de un archivo será la primera operación a realizar. Una vez que el archivo ha sido creado, la mayoría de los usuarios simplemente desearán acceder al archivo y a la información contenida en él.

Para crear un nuevo archivo dentro de un sistema de computadora se necesitan los siguientes datos:

- *Nombre dispositivo*: indica el lugar donde se situará el archivo cuando se cree.
- *Nombre del archivo*: identifica el archivo entre los restantes archivos de una computadora.
- *Tamaño del archivo*: indica el espacio necesario para la creación del archivo.
- *Organización del archivo*: tipo de organización del archivo.
- *Tamaño del bloque o registro físico*: cantidad de datos que se leen o escriben en cada operación de entrada/salida (E/S).

Al ejecutar la creación de un archivo se puede generar una *serie de errores*, entre los que se pueden destacar los siguientes:

- Otro archivo con el mismo nombre ya existía en el soporte.
- El dispositivo no tiene espacio disponible para crear otro nuevo archivo.
- El dispositivo no está operacional.
- Existe un problema de hardware que hace abortar el proceso.
- Uno o más de los parámetros de entrada en la instrucción son erróneos.

La instrucción o acción en pseudocódigo que permite crear un archivo se codifica con la palabra **crear**.

```
crear(<var_tipo_archivo>, <nombre_físico>)
```

9.6.2. Abrir un archivo

La acción de *abrir (open)* un archivo es permitir al usuario localizar y acceder a los archivos que fueron creados anteriormente.

La diferencia esencial entre una instrucción de *abrir* un archivo y la de *crear* un archivo residen en que el archivo no existe antes de utilizar **crear** y se supone que debe existir antes de utilizar **abrir**.

La información que un sistema de tratamiento de archivos requiere para abrir un archivo es diferente de las listas de información requerida para crearlo. La razón para ello reside en el hecho que toda la información que

Krasher Podres 1995564-2202180125

realmente describe el archivo se escribió en éste durante su proceso de creación. Por consiguiente, la operación **crear** sólo necesita localizar y leer esta información conocida como atributos del archivo.

La instrucción de abrir un archivo consiste en la creación de un canal que comunica a un usuario a través de un programa con el archivo correspondiente situado en un soporte.

Los parámetros que se deben incluir en una instrucción de apertura (**abrir**) son:

- Nombre del dispositivo.
- Nombre del usuario o canal de comunicación.
- Nombre del archivo.

Al ejecutar la instrucción **abrir** se pueden encontrar los siguientes errores:

- Archivo no encontrado en el dispositivo especificado (nombre de archivo o identificador de dispositivo erróneo).
- Archivo ya está en uso para alguna otra aplicación del usuario.
- Errores hardware.

El formato de la instrucción es:

```
Abrir (<var_tipo_archivo>,<modo>,<nombre_físico>)
```

La operación de abrir archivos se puede aplicar para operaciones de lectura (l), escritura (e), lectura/escritura (l/e).

abrir (*id_archivo, l, nombre_archivo*)

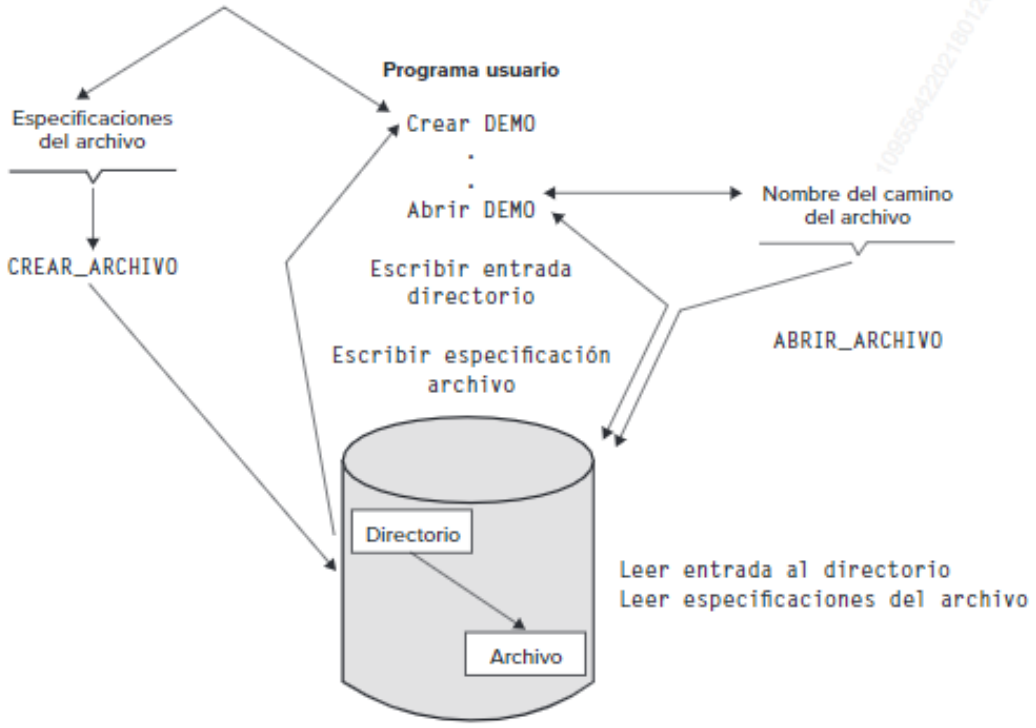


Figura 9.16. Abrir un archivo.

Como ya se mencionó, para que un archivo pueda abrirse ha de haber sido previamente creado. Cuando un archivo se abre para lectura colocamos un hipotético puntero en el primer registro del archivo y se permitirán únicamente operaciones de lectura de sus registros. La apertura para escritura coloca dicho hipotético puntero detrás del

Krasher Padres 1095564-2202180126

último registro del archivo, y dispuesto para la adición de nuevos registros en él. Ambos modos se consideran propios de archivos secuenciales. Los archivos directos se abrirán en modo lectura/escritura, permitiéndose tanto la lectura como la escritura de nuevos registros.

9.6.3. Cerrar archivos

El propósito de la operación de cerrar un archivo es permitir al usuario cortar el acceso o detener el uso del archivo, permitiendo a otros usuarios acceder a él. Para ejecutar esta función, el sistema de tratamiento de archivos sólo necesita conocer el nombre del archivo que se debe cerrar, y que previamente debía estar abierto.

Formato:

```
cerrar (<var_tipo-archivo>
```

Estructura:

9.6.4. Borrar archivos

La instrucción `borrar` tiene como objetivo la supresión de un archivo del soporte o dispositivo. El espacio utilizado por un archivo borrado puede ser utilizado por otros.

La información necesaria para eliminar un archivo es:

- Nombre del dispositivo y número del canal de comunicación.
- Nombre del archivo.

Los *errores* que se pueden producir son:

- El archivo no se puede encontrar bien porque el nombre no es válido o porque nunca existió.
- Otros usuarios estaban actuando sobre el archivo y estaba activo.
- Se detectó un problema de hardware.

Krasner Padres 1095584-2202180126

1095584220218012

9.7. Flujos

Un **archivo** o **fichero** es una colección de datos relacionados. En esencia, C++ o Java visualizan cada archivo como un **flujo** (*stream*) secuencial de bytes. En la entrada, un programa extrae bytes de un flujo de entrada y en la salida, inserta bytes en el flujo de salida. En un programa orientado a texto, cada byte representa un carácter; en general, los bytes pueden formar una representación binaria de datos carácter o numéricos. Los bytes de un flujo de entrada pueden venir del teclado o de un escáner, por ejemplo, pero también pueden venir de un dispositivo de almacenamiento, como un disco duro o un CD, o desde otro programa. De modo similar, en un flujo de salida, los bytes pueden fluir a la pantalla, a una impresora, a un dispositivo de almacenamiento o a otro programa. En resumen, un flujo actúa como un intermediario entre el programa y el destino o fuente del flujo.

Este enfoque permite a un programa C++, Java,..., tratar la entrada desde un archivo. En realidad el lenguaje trata un archivo como una serie de bytes; muchos archivos residen en un disco, pero dispositivos como impresoras, discos magnéticos y ópticos, y líneas de comunicación se consideran archivos. Con este enfoque, por ejemplo, un programa C++ examina el flujo de bytes sin necesidad de conocer su procedencia, y puede procesar la salida de modo independiente adonde vayan los bytes.

Un archivo es un flujo secuencial de bytes. Cada archivo termina con una marca final de archivo (**EOF**, *end-of-file*) o en un número de byte específico grabado en el sistema. Un programa que procesa un flujo de byte recibe una indicación del sistema cuando se alcanza el final del flujo con independencia de cómo estén representados los flujos o archivos.

9.7.1. Tipos de flujos

Existen dos tipos de flujos en función del sentido del canal de comunicación: **flujo de entrada** y **flujo de salida**. El primero lee información como una secuencia de caracteres; éstos pueden ser tecleados en la consola de entrada,

leídos de un archivo de entrada, o leídos de zócalo de una red. Un **flujo de salida** es una secuencia de caracteres que se almacenan como información, éstos se pueden visualizar en la consola, escribir en un archivo de salida o en zócalos de red.

Un *flujo* de entrada envía datos desde una fuente a un programa. Un flujo de *salida* envía datos desde un programa a un destino.

Desde el punto de vista de la información que contienen, los flujos se clasifican en:

- *Flujos de bytes*, se utilizan para manejar bytes, enteros y otros tipos de datos simples. Se pueden expresar en formato bytes, incluyendo datos numéricos, programas ejecutables, comunicaciones de Internet, *bytecode* (archivos de clases ejecutados por una máquina virtual Java). Cada tipo de dato se puede expresar como bytes individuales o bien como combinación de bytes.
- *Flujos de caracteres*, manipulan archivos de texto y otras fuentes de texto. Se diferencian de los flujos de bytes en que soportan el conjunto de caracteres ASCII o Unicode. Cualquier tipo de datos que implique texto debe utilizar flujo de caracteres, incluyendo archivos de texto, páginas web o sitios comunes de texto.

9.7.2. Flujos en C++

La gestión de la entrada implica dos etapas:

- Asociación de un flujo con una entrada a un programa.
- Conexión del flujo a un archivo.

En otras palabras, un flujo de entrada necesita dos conexiones, una en cada extremo. La conexión fin de archivo proporciona una fuente para el flujo y la conexión fin de programa vuelca el flujo de salida al programa (la conexión final de archivo, también puede ser un dispositivo, como un teclado). De igual modo, la gestión de salida implica la conexión de un flujo de salida al programa y la asociación de un destino de salida con el flujo. Al igual que sucede en una tubería del servicio del agua corriente de su ciudad, fluyen bytes en lugar de agua.

En C++ un flujo es un tipo especial de variable conocida como un objeto. Los flujos `cin` y `cout` se utilizan en entradas y salidas. La clase istream define el operador de extracción (>>) para los tipos primitivos. Este operador convierte los datos a una secuencia de caracteres y los inserta en el flujo.

Los flujos cin y cout se declaran en el lenguaje por usted, pero si desea que un flujo se conecte a un archivo, se debe declarar justo antes de que se pueda declarar cualquier otra variable.

9.7.3. Flujos en Java

El procedimiento para utilizar bien un flujo de bytes o un flujo de caracteres en Java es, en gran medida, el mismo. Antes de comenzar a trabajar con las clases específicas de la biblioteca de clases java.io, es útil revisar el proceso de crear y utilizar flujos.

Para un flujo de entrada, el primer paso es crear un objeto asociado con la fuente de datos. Por ejemplo, si la fuente es un archivo de su unidad de disco duro, un objeto FileInputStream se puede asociar con este archivo.

Después que se tiene un objeto de flujo, se puede leer la información desde el flujo utilizando uno de los métodos del objeto `FileInputStream` incluye un método `read` que devuelve un byte leído desde el teclado.

Cuando se termina de leer la información del flujo se llama al método `close( )` para indicar que se ha terminado de utilizar el flujo.

En el caso de un flujo de salida, se crea un objeto asociado con el destino de los datos. Tal objeto se puede crear de la clase `BufferedWriter` que representa un medio eficiente de crear archivos de texto.

El método `write( )` es el medio más simple para enviar información al destino del flujo de salida. Al igual que con los flujos de entrada, el método `close( )` se llama en un flujo de salida cuando no se tiene más información que enviar.

Krasner Padres 1095564-2202180127

9.7.4. Consideraciones prácticas en Java y C#

Java y C# realizan las operaciones en archivos a través de flujos, manipulados por clases, que conectan con el medio de almacenamiento. De esta forma, para crear y abrir un archivo se requiere utilizar una clase que defina la funcionalidad del flujo, el cual determina el sentido de la comunicación (lectura, escritura, o lectura/escritura), la posibilidad de posicionamiento directo o no en un determinado registro y la forma de leer y/o escribir en el archivo. Cerrar el archivo implica cerrar el flujo. Así la siguiente instrucción en Java crea un flujo que permite la lectura/escritura (`rw`) en un archivo donde se podrá efectuar posicionamiento directo y cuyo nombre externo es `empleados.dat`.

```
RandomAccessFile e = new RandomAccessFile ("empleados.dat", "rw");
```

Pueden utilizarse flujos de bytes, caracteres, cadenas o tipos primitivos. Por ejemplo, en Java la clase `FileInputStream` permite crear un flujo para lectura secuencial de bytes desde un archivo, mientras `FileReader` lo crea para la lectura secuencial de caracteres y `RandomAccessFile`, como ya se ha comentado, admite posicionamiento directo y permite la lectura/escritura de datos tipos primitivos.

La personalización de flujos se consigue por asociación o encadenamiento de otros sobre los flujos base de apertura de archivos. Una aplicación práctica de esta propiedad en Java puede ser permitir la lectura de una cadena de caracteres desde un flujo de entrada.

```
BufferedReader f = new BufferedReader (new FileReader("datos.txt"));
cadena = f.readLine();      //lee una cadena del archivo
f.close();                  // cierra el archivo
```

En C# la situación es similar y sobre los flujos base, que conectan al medio de almacenamiento, pueden encadenarse otros para efectuar tratamientos especiales de la información.

```
BinaryWriter f = new BinaryWriter (new FileStream("notas.dat",
                 FileMode.OpenOrCreate, FileAccess.Write));
/* BinaryWriter proporciona métodos para escribir tipos de
 datos primitivos en formato binario */

f.Write (5.34 * 2);

f.Close();      // Cerrar el archivo
```

9.8. Mantenimiento de archivos

La operación de mantenimiento de un archivo incluye todas las operaciones que sufre un archivo durante su vida y desde su creación hasta su eliminación o borrado.

El mantenimiento de un archivo consta de dos operaciones diferentes:

- *Actualización.*
- *Consulta.*

La *actualización* es la operación de eliminar o modificar los datos ya existentes, o bien introducir nuevos datos. En esencia, es la puesta al día de los datos del archivo.

Las operaciones de actualización son:

- *Altas.*
- *Bajas.*
- *Modificaciones.*

Las operaciones de consulta tienen como finalidad obtener información total o parcial de los datos almacenados en un archivo y presentarlos en dispositivos de salida: pantalla o impresora, bien como resultados o como listados.

Todas las operaciones de mantenimiento de archivos suelen constituir módulos independientes del programa principal y su diseño se realiza con subprogramas (*subrutinas* o *procedimientos* específicos).

Así, los subprogramas de mantenimiento de un archivo constarán de:

Krasner Podres 1095564-2202180128

Altas

Una operación de *alta* en un archivo consiste en la adición de un nuevo registro. En un archivo de empleados, un alta consistirá en introducir los datos de un nuevo empleado. Para situar correctamente un alta, se deberá conocer la posición donde se desea almacenar el registro correspondiente: al principio, en el interior o al final de un archivo.

El algoritmo del subprograma `ALTAS` debe contemplar la comprobación de que el registro a dar de alta no existe previamente.

Bajas

Una *baja* es la acción de eliminar un registro de un archivo. La baja de un registro se puede presentar de dos formas: indicación del registro específico que se desea dar de baja o bien visualizar los registros del archivo para que el usuario elija cuál borrará.

La baja de un registro puede ser *lógica* o *física*. Una *baja lógica* supone el no borrado del registro en el archivo. Esta baja lógica se manifiesta en un determinado campo del registro con una *bandera, indicador* o *flag* –carácter *, $, etcétera–, o bien con la escritura o rellenado con espacios en blanco de algún campo en el registro específico.

Una *baja física* implica el borrado y desaparición del registro, de modo que se crea un nuevo archivo que no incluye el registro dado de baja.

Modificaciones

Una *modificación* en un archivo consiste en la operación de cambiar total o parcialmente el contenido de uno de sus registros. Esta fase es típica cuando cambia el contenido de un determinado campo de un archivo; por ejemplo, la dirección o la edad de un empleado.

La forma práctica de modificar un registro es la visualización del contenido de sus campos; para ello se debe elegir el registro o registros a modificar. El proceso consiste en la lectura del registro, modificación de su contenido y escritura total o parcial del mismo.

Consulta

La operación de *consulta* tiene como fin visualizar la información contenida en el archivo, bien de un modo completo –bien de modo parcial–, examen de uno o más registros.

Las operaciones de consulta de archivo deben contemplar diversos aspectos que faciliten la posibilidad de conservación de datos. Los aspectos más interesantes a tener en cuenta son:

- Opción de visualización en pantalla o listado en impresora.
- Detención de la consulta a voluntad del usuario.
- Listado por registros o campos individuales o bien listado total del archivo (en este caso deberá existir la posibilidad de impresión de listados, con opciones de saltos de página correctos).

9.8.1. Operaciones sobre registros

Las operaciones de transferencia de datos a/desde un dispositivo a la memoria central se realizan mediante las instrucciones:

```
leer (<var_tipo_archivo>, lista de entrada de datos)
escribir (<var_tipo_archivo>, lista de salida de datos)
```

organización directa

```
lista de entrada de datos = numero_registro, nombre_registro
lista de salida de datos = numero_registro, nombre_registro
```

organización secuencial

```
lista de entrada de datos = <lista_de_variables>
lista de salida de datos = <lista_de_expresiones>
```

Las operaciones de acceso a un registro y de paso de un registro a otro se realiza con las acciones **`leer`** y **`escribir`**.

Krasner Podres 1095564-2202180129

9.9. Procesamiento de archivos secuenciales (algoritmos)

En un archivo secuencial los registros se insertan en el archivo en orden cronológico de llegada al soporte, es decir, un registro de datos se almacena inmediatamente a continuación del anterior. Los archivos secuenciales terminan con una marca final de archivo (FDA o EOF), cuando se tengan que añadir registros se añadirán al final, inmediatamente por delante de las marcas fin de archivos.

Las operaciones básicas que se permiten en un archivo secuencial son: *escribir su contenido, añadir un registro al final del archivo* y *consultar sus registros*. Las demás operaciones exigen una programación específica.

Los archivos secuenciales son los que ocupan menos memoria y son útiles cuando se desconoce *a priori* el tamaño de los datos y se requieren registros de longitud variable. También son muy empleados para el almacenamiento de información, cuyos contenidos sufran pocas modificaciones en el transcurso de su vida útil.

Es característico de los archivos secuenciales el no poder ser utilizados simultáneamente para lectura y escritura.

9.9.1. Creación

La *creación* de un archivo secuencial es un proceso secuencial, ya que los registros se almacenan consecutivamente en el mismo orden en que se introducen en el archivo. El método de creación de un archivo consiste en la ejecución de un programa adecuado que permita la entrada de datos al archivo desde el terminal. El sistema usual es el *interactivo*, en el que el programa solicita los datos al usuario que los introduce por teclado, al terminar se introduce una marca final de archivo, que supone el final físico del archivo.

En los archivos secuenciales, EOF o FDA es una función lógica que toma el valor *cierto* si se ha alcanzado el final de archivo y *falso* en caso contrario.

La creación del archivo requerirá los siguientes pasos:

- Abrir el archivo.
- Leer datos del registro.
- Grabar registro.
- Cerrar archivo.

El algoritmo de creación es el siguiente:

```
algoritmo  crea_sec
tipo
   registro: datos_personales
      <tipo_dato1>: nombre_campo1
      <tipo_dato2>: nombre_campo2
      ............................
   fin_registro
   archivo_s de datos_personales: arch
var
   arch                :f
   datos_personales    :persona
inicio
   crear (f,<nombre_en_disco>)
   abrir (f, e,<nombre_en_disco>)
   leer_reg (persona)
   { utilizamos un procedimiento para no tener que detallar la lectura}
   mientras no último_dato(persona) hacer
      escribir_f_reg (f, persona)
   //la escritura se realizará campo a campo
      leer_reg(persona)
   fin_mientras
   cerrar(f)
fin
```

Krasher Padres 1095564-2202180130

Se considera que se permite la lectura y escritura en el archivo de los datos, tal y como se almacenan en memoria. Un archivo de texto es secuencial en el que sólo se leen y escriben series de caracteres y no sería necesario especificar en la declaración del archivo el tipo de registros que lo constituyen, pues siempre son líneas.

9.9.2. Consulta

El proceso de búsqueda o consulta de una información en un archivo de organización secuencial se debe efectuar obligatoriamente en modo secuencial. Por ejemplo, si se desea consultar la información contenida en el registro 50, se deberán leer previamente los 49 primeros registros que le preceden en orden secuencial. En el caso de un archivo de personal, si se desea buscar un registro determinado correspondiente a un empleado en particular, será necesario recorrer —leer— todo el archivo desde el principio hasta encontrar el registro que se busca o la marca final de archivos.

Así, para el caso de un archivo de n registros, el número de lecturas de registros efectuadas son:

- Mínimo 1, si el registro buscado es el primero del archivo.
- Máximo n, si el registro buscado es el último o no existe dentro del archivo.

Por término medio, el número de lecturas necesarias para encontrar un determinado registro es:

$$\frac{n+1}{2}$$

El tiempo de acceso será influyente en las operaciones de lectura/escritura. Así, en el caso de una lista o vector de n elementos almacenados en memoria central puede suponer tiempos de microsegundos o nanosegundos; sin embargo, en el caso de un archivo de n registros los tiempos de acceso son de milisegundos o fracciones/múltiples de segundos, lo que supone un tiempo de acceso de 1000 a 100000 veces más grande una búsqueda de información en un soporte externo que en memoria central.

El algoritmo de consulta de un archivo requerirá un diseño previo de la presentación de la estructura de registros en el dispositivo de salida, de acuerdo con el número y longitud de los campos.

```
algoritmo consulta_sec
tipo
   registro: datos_personales
      <tipo_dato1>: nombre_campo1
      <tipo_dato2>: nombre_campo2
      ..............: ..............
   fin_registro
   archivo_s de datos_personales: arch
var
   arch: f
   datos_personales:     persona
inicio
   abrir(f, l, <nombre_en_disco>)
   mientras no fda(f)hacer
      leer_f_reg(f, persona)
   fin_mientras
   cerrar(f)
fin
```

o bien:

```
inicio
   abrir(f, l, <nombre_en_disco>)
   leer_f_reg(f, persona)
```

Krasner Padres 1095584-2202160131

```
   mientras no fda(f) hacer
      escribir_reg(persona)
      leer_f_reg(f, persona)
   fin_mientras
   cerrar(f)
fin
```

El uso de uno u otro algoritmo depende de cómo el lenguaje de programación detecta la marca de fin de archivo. En la mayoría de los casos el algoritmo válido es el primero, pues la marca se detecta automáticamente con la lectura del último registro.

En el caso de búsqueda de un determinado registro, con un campo clave *x*, el algoritmo de búsqueda se puede modificar en la siguiente forma con:

Consulta de un registro

Si el archivo no está ordenado:

```
algoritmo consultal_sec
tipo
   registro: datos_personales
      <tipo_dato1>:nombre_campo1
      <tipo_dato2>:nombre_campo2
      ..... : .....
   fin_registro
   archivo_s de datos_personales: arch
var
   arch              :f
   datos_personales:persona
   <tipo_dato1>      :clavebus
   lógico            :encontrado
inicio
   abrir(f, l, <nombre en_disco>)
   encontrado ← falso
   leer(clavebus)
   mientras no encontrado y no fda(f) hacer
      leer_f_reg(f, persona)
      si igual(clavebus, persona) entonces
         encontrado ← verdad
      fin_si
   fin_mientras
   si no encontrado entonces
      escribir ('No existe')
   si_no
      escribir_reg(persona)
   fin_si
   cerrar(f)
fin
```

Si el archivo está indexado en orden creciente por el campo por el cual realizamos la búsqueda, el proceso se podría acelerar haciendo innecesario recorrer todo el fichero para averiguar que un determinado registro no está:

```
algoritmo consulta2_sec
tipo
   registro: datos_personales
      <tipo_dato1>: nombre_campo1
```

Krasnar Padres 1095564-2202180131
1095564220218013

```
      <tipo_dato2>: nombre_campo2
      ............: ............
   fin_registro
   archivo_s de datos_personales: arch
var
   arch            : f
   datos_personales: persona
   <tipo_dato1>    : clavebus
   lógico          : encontrado, pasado
inicio
   abrir(f, l, <nombre_en_disco>)
   encontrado ← falso
   pasado ← falso
   leer(clavebus)
   mientras no encontrado y no pasado y no fda(f) hacer
        leer_f_reg(f, persona)
      si igual(clavebus, persona) entonces
         encontrado ← verdad
      si_no
         si menor(clavebus, persona) entonces
            pasado ← verdad
         fin_si
      fin_si
   fin_mientras
si no encontrado entonces
   escribir ('No existe')
   si_no
      escribir_reg(persona)
   fin_si
   cerrar(f)
fin
```

9.9.3. Actualización

La actualización de un archivo supone:

- Añadir nuevos registros (*altas*).
- Modificar registros ya existentes (*modificaciones*).
- Borrar registros (*bajas*).

Altas

La operación para dar de alta un determinado registro es similar a la operación de añadir datos a un archivo.

```
algoritmo añade_sec
tipo
   registro: datos_personales
      <tipo_dato1>: nombre_campo1
      <tipo_dato2>: nombre_campo2
      ............:............
   fin_registro
   archivo_s de datos_personales:arch
```

Krasner Padres 1095564-2202180132

```
var
   arch            : f
   datos_personales: persona
inicio
   abrir(f, e, <nombre_en_disco>)
   leer_reg(persona)
   mientras no último_dato(persona) hacer
      escribir_f_reg (f, persona)
      leer_reg (persona)
   fin_mientras
   cerrar
fin
```

Bajas

Existen dos métodos para dar de baja un registro:

1. Se utiliza un archivo transitorio.
2. Se almacenan en un *array* (vector) todos los registros del archivo, señalando con un indicador o bandera (*flag*) el registro que se desea dar de baja.

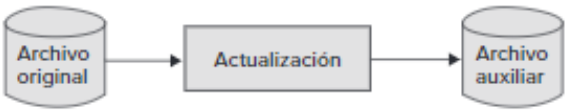
Krasher Padres 1995564-2202180133

Método 1

Se crea un segundo archivo auxiliar, también secuencial, copia del que se trata de actualizar. Se lee el archivo completo registro a registro y en función de su lectura se decide si se debe dar de baja o no.

Si el registro se va a dar de baja, se omite la escritura en el archivo auxiliar o transitorio; si no, este registro se escribe en el archivo auxiliar.

Tras terminar la lectura del archivo original, se tendrán dos archivos: *original* (o maestro) y *auxiliar*.



El proceso de bajas del archivo concluye cambiando el nombre del archivo auxiliar por el maestro, borrando previamente el archivo maestro original.

```
algoritmo bajas_s
tipo
   registro: datos_personales
      <tipo_dato1>: nombre_campo1
      <tipo_dato2>: nombre_campo2
      ..............:...............
   fin_registro
   archivo_s de datos_personales:arch
var
   arch            :f, faux
   datos_personales: persona, personaaux
   lógico          :encontrado
inicio
   abrir(f, l, 'antiguo')
   crear(faux, 'nuevo')
   abrir(faux, e, 'nuevo')
   leer(personaaux.nombre_campo1)
```

```
   encontrado ← falso
   mientras no fda (f) hacer
      leer_f_reg (f, persona)
      si personaaux.nombre_campo1 = persona.nombre_campo1 entonces
         encontrado ← verdad
      si_no
         escribir_f_reg (faux, persona)
      fin_si
fin_mientras
   si no encontrado entonces
      escribir ('No está')
   fin_si
   cerrar (f, faux)
   borrar ('antiguo')
   renombrar ('nuevo', 'antiguo')
fin
```

Método 2

Este procedimiento consiste en señalar los registros que se desean dar de baja con un indicador o bandera, creando así un nuevo archivo secuencial sin los mismos.

Modificaciones

El proceso de modificación de un registro consiste en localizar este registro, efectuar dicha modificación y a continuación reescribir el nuevo registro en el archivo. El proceso es similar al de bajas:

```
algoritmo modificación_sec
tipo
   registro: datos_personales
      <tipo_dato1>: nombre_campo1
      <tipo_dato2>: nombre_campo2
      ..............:..............
   fin
   archivo_s de datos_personales: arch
var
   arch            : f, faux
   datos_personales: persona, personaaux
   lógico          : encontrado
inicio
   abrir(f, l, 'antiguo')
   crear(faux, 'nuevo')
   abrir(faux, e, 'nuevo')
   leer(personaaux.nombre_campo1)
   encontrado ← falso
   mientras_no fda(f) hacer
      leer_f_reg (f, persona)
      si personaaux.nombre_campo1=persona.nombre_campo1 entonces
         encontrado ← verdad
         modificar (persona)
      fin_si
      escribir_f_reg (faux, persona)
   fin_mientras
```

Krasner Padres 1095564-2202180134

```
      si no encontrado entonces
                  escribir ('No está')
      fin_si
      cerrar(f, faux)
      borrar('antiguo')
      renombrar ('nuevo', 'antiguo')
   fin
```

El subprograma de modificación de su registro consta de unas pocas instrucciones en las que se debe introducir por teclado el registro completo con indicación de todos sus campos o, por el contrario, el campo o campos que se desea modificar. El subprograma en cuestión podría ser:

```
procedimiento modificar(E/S datos_personales: persona)
var carácter: opción
   entero   : n
inicio
   escribir('R.- registro completo')
   escribir('C.- campos individuales')
   escribir('elija opción:')
   leer(opción)
según_sea opción hacer
   'R'
      visualizar(persona)
      leer_reg(persona)
   'C'
      presentar(persona)
      solicitar_campo(n)
      introducir_campo(n, persona)
   fin_según
fin_procedimiento
```

Krasner Podres 1095584-2202180135

1095584220218013S

9.10. Procesamiento de archivos directos (algoritmos)

Se dice que un archivo es aleatorio o directo cuando cualquier registro es directamente accesible mediante la especificación de un índice, el cual da la posición del registro con respecto al origen del fichero. Los archivos aleatorios o directos tienen una gran rapidez para el acceso comparado con los secuenciales; los registros son fáciles de referenciar —número de orden del registro—, lo que representa una gran facilidad de mantenimiento.

La lectura/escritura de un registro es rápida, ya que se accede directamente al registro y no se necesita recorrer los anteriores.

9.10.1. Operaciones con archivos directos

Las operaciones con archivos directos son las usuales.

Creación

El proceso de creación de un archivo directo o aleatorio consiste en ir introduciendo los sucesivos registros en el soporte que los va a contener y en la dirección obtenida, resultante del algoritmo de conversión. Si al introducir un registro se encuentra ocupada la dirección, el nuevo registro deberá ir a la zona de sinónimos o de excedentes.

```
algoritmo crea_dir
   tipo
      registro: datos_personales
         <tipo_dato1> : nombre_campo1
         ..........  : ............
         <tipo_datoN> : nombre_campoN
         ..........  : ............
   fin_registro
   archivo_d de datos_personales: arch
var
   arch             : f
   datos_personales : persona
inicio
   crear(f,<nombre_en_disco>)
   abrir(f, l/e, <nombre_en_disco>)
   ........................
   { las operaciones pueden variar con arreglo al modo como
   pensemos trabajar posteriormente con el archivo
   (posicionamiento directo en un determinado registro,
   transformación de clave, indexación) }
   ........................
   cerrar(f)
fin
```

En los registros de un archivo directo se suele incluir un campo –ocupado– que pueda servir para distinguir un registro dado de baja o modificado de un alta o de otro que nunca contuvo información.

Dentro del proceso de creación del archivo podríamos considerar una inicialización de dicho campo en cada uno de los registros del archivo directo.

```
algoritmo crea_dir
const
      max = <valor>
tipo
   registro: datos_personales
      <tipo_dato1>: cod
      <tipo_dato2>: ocupado
      ..........  : ............
      <tipo_daton>: nombre_campon
      ..........  : ............
   fin_registro
   archivo_d de datos_personales: arch
var
   arch             : f
   datos_personales : persona
inicio
   crear(f, <nombre_en_disco>)
   abrir(f, l/e, <nombre_en_disco>)
   desde i ← 1 hasta Max hacer
      persona.ocupado ← ''
      escribir(f, i, persona)
   fin_desde
   cerrar(f)
fin
```

Krasher Padres 1095564-2202180136

Altas

La operación de altas en un archivo directo o aleatorio consiste en ir introduciendo los sucesivos registros en una determinada posición, especificada a través del índice. Mediante este índice nos posicionaremos directamente sobre el byte del fichero que se encuentra en la posición `(indice - 1) * tamaño_de(<tipo_registros_del_archivo>)` y escribiremos allí nuestro registro.

Tratamiento por transformación de clave

El método de transformación de clave consiste en transformar un número de orden (clave) en direcciones de almacenamiento por medio de un algoritmo de conversión.

Cuando las altas se realizan por el método de transformación de clave, la dirección donde introducir un determinado registro se conseguirá por la aplicación a la clave del algoritmo de conversión (HASH). Si encontráramos que dicha dirección ya está ocupada, el nuevo registro deberá ir a la zona de sinónimos o de excedentes.

```
algoritmo altas_dir_trcl
const
   findatos = <valor1>
   max      = <valor2>
tipo
   registro: datos_personales
      <tipo_dato1>: cod
      <tipo_dato2>: ocupado
      ..... : .....
      <tipo_daton>: nombre_campon
      ..... : .....
   fin_registro
   archivo_d de datos_personales: arch
var
   arch                 : f
   datos_personales     : persona, personaaux
   lógico               : encontradohueco
   entero               : posi
inicio
   abrir(f, l/e, <nombre_en_disco>)
   leer(personaaux.cod)
   posi ← HASH(personaaux.cod)
   leer(f, posi, persona)
   si persona.ocupado = '*' entonces
      encontradohueco ← falso
      posi ← findatos
      mientras posi < Máx y no encontradohueco hacer
         posi ← posi + 1
         leer(f, posi, persona)
         si persona.ocupado <> '*' entonces
            encontradohueco ← verdad
         fin_si
      fin_mientras
   si_no
      encontradohueco ← verdad
   fin_si
   si encontradohueco entonces
      leer_otros_campos(personaaux)
      persona ← personaaux
      persona.ocupado ← '*'
```

Krasher Padres 1095564-2202180137

```
      escribir(f, posi, persona)
   si_no
      escribir('No está')
   fin_si
   cerrar(f)
fin
```

Consulta

El proceso de consulta de un archivo directo o aleatorio es rápido y debe comenzar con la entrada del índice correspondiente al registro que deseamos consultar.

El índice permitirá el posicionamiento directo sobre el byte del fichero que se encuentra en la posición

```
(indice - 1) * tamaño_de(<var_de_tipo_registros_del_fichero>)
```

```
algoritmo consultas_dir
const
   max     = <valor1>
tipo
   registro: datos_personales
   { Cuando el código coincide con el índice o posición del
   registro en el archivo, no resulta necesario su
   almacenamiento }
      <tipo_dato1>: ocupado
      ..........  : ............
      <tipo_daton>: nombre_campon
      ..........  : ............
   fin_registro
   archivo_d de datos_personales: arch
var
   arch              : f
   datos_personales  : persona
   lógico            : encontrado
   entero            : posi
inicio
abrir(f, l/e, <nombre_en_disco>)
leer(posi)
si (posi >=1) y (posi <= Max) entonces
   leer(f, posi, persona)
   {como al escribir los datos marcamos el campo
   ocupado con * }
   si persona.ocupado <> "*" entonces
      {para tener garantías en esta operación es
      por lo que debemos inicializar en todos los
      registros, durante el proceso de creación, el
      campo ocupado a un determinado valor,
      distinto de *}
      encontrado ← falso
   si_no
      encontrado ← verdad
   fin_si
   si encontrado entonces
      escribir_reg(persona)
   si_no
```

Krasher Padres 1095564-2202180137
1095564-2202180137

```
      escribir('No está')
   fin_si
si_no
      escribir('Número de registro incorrecto')
fin_si
cerrar(f)
fin
```

Consulta. *Por transformación de clave.*

Puede ocurrir que la clave o código por el que deseamos acceder a un determinado registro no coincida con la posición de dicho registro en el archivo, aunque guarden entre sí una cierta relación, pues al escribir los registros en el archivo la posición se obtuvo aplicando a la clave un algoritmo de conversión.

En este caso es imprescindible el almacenamiento de la clave en uno de los campos del registro, y las operaciones a realizar para llevar a cabo una consulta serían:

- Definir clave del registro buscado.
- Aplicar algoritmo de conversión clave a dirección.
- Lectura del registro ubicado en la dirección obtenida.
- Comparación de las claves de los registros leído y buscado y, si son distintas, exploración secuencial del área de excedentes.
- Si tampoco se encuentra el registro en esta área es que no existe.

Krasher Padres 1095564-2202180138

```
algoritmo consultas_dir_trcl
const
   findatos = <valor1>
   max      = <valor2>
tipo
   registro: datos_personales
      <tipo_dato1>: cod
      <tipo_dato2>: ocupado
      ..... : .....
      <tipo_daton>: nombre_campon
      ..... : .....
   fin_registro
   archivo_d de datos_personales: arch
var
   arch               : f
   datos_personales   : persona, personaaux
   lógico             : encontrado
   entero             : posi
inicio
abrir(f, l/e, <nombre_en_disco>)
leer(personaaux.cod)
posi ← HASH(personaaux.cod)
leer(f, posi, persona)
si (persona.ocupado <> '*') o (persona.cod <> personaaux.cod) entonces
   encontrado ← falso
   posi ← Findatos
   mientras (posi < Max ) y no encontrado hacer
      posi ← posi + 1
      leer(f, posi, persona)
      si (persona.ocupado = '*' )y
         (persona.cod = personaaux.cod) entonces
         encontrado ← verdad
      fin_si
```

```
      fin_mientras
   si_no
      encontrado ← verdad
   fin_si
   si encontrado entonces
      escribir_reg(persona)
   si_no
      escribir('No está')
   fin_si
   cerrar(f)
   fin
```

Bajas

En el proceso de bajas se considera el contenido de un campo indicador, por ejemplo, `persona.ocupado`, que, cuando existe información válida en el registro está marcado con un *. Para dar de baja al registro, es decir, considerar su información como no válida, eliminaremos dicho *. Este tipo de baja es una baja lógica.

Desarrollaremos a continuación un algoritmo que realice bajas lógicas y acceda a los registros a los que se desea dar la baja por el método de transformación de clave.

```
algoritmo bajas_dir_trcl
const
   findatos = <valor1>
   max      = <valor2>
tipo
   registro: datos_personales
      <tipo_dato1>: cod
      <tipo_dato2>: ocupado
      ..... : .....
      <tipo_daton>: nombre_campon
      ..... : .....
   fin_registro
   archivo_d de datos_personales: arch
var
   arch             : f
   datos_personales : persona, personaaux
   lógico           : encontrado
   entero           : posi
inicio
abrir(f,l/e, <nombre_en_disco>)
leer(personaaux.cod)
posi ← HASH(personaaux.cod)
leer(f, posi, persona)
si (persona.ocupado <> '*') o
   (persona.cod <> personaaux.cod) entonces
   encontrado ← falso
   posi ← findatos
   mientras (posi < Max) y no encontrado hacer
      posi ← posi + 1
      leer(f, posi, persona)
      si (persona.ocupado = '*') y
         (persona.cod = personaaux.cod) entonces
        encontrado ← verdad
```

Krasher Podres 1095584-2202180139

```
      fin_si
   fin_mientras
si_no
   encontrado ← verdad
fin_si
si encontrado entonces
   persona.ocupado ← ' '
   escribir(f, posi, persona)
si_no
   escribir('No está')
fin_si
   cerrar(f)
fin
```

Modificaciones

En un archivo aleatorio se localiza el registro que se desea modificar —mediante la especificación del índice o aplicando el algoritmo de conversión clave a dirección y, en caso necesario, la búsqueda en la zona de colisiones— se modifica el contenido y se reescribe.

```
algoritmo modificaciones_dir_trcl
const
   findatos = <valor1>
   max      = <valor2>
tipo
   registro: datos_personales
      <tipo_dato1>: cod
      <tipo_dato2>: ocupado
      ..... : .....
      <tipo_daton>: nombre_campon
      ..... : .....
   fin_registro
   archivo_d de datos_personales: arch
var
   arch             : f
   datos_personales : persona, personaaux
   lógico           : encontrado
   entero           : posi
inicio
   abrir(f,l/e, <nombre_en_disco>)
   leer(personaaux.cod)
   posi ← HASH(personaaux.cod)
   leer(f, posi, persona)
   if (persona.ocupado <> '*') o
      (persona.cod <> personaaux.cod) entonces
      encontrado ← falso
      posi ← findatos
      mientras posi < max y no encontrado hacer
         posi ← posi + 1
         leer(f, posi, persona)
         si (persona.ocupado = '*') y (persona.cod = personaaux.cod)
            entonces
           encontrado ← verdad
         fin_si
```

Krasner Podres 1095584-2202180140
1095584220218014O

```
      fin_mientras
   si_no
      encontrado ← verdad
   fin_si
   si encontrado entonces
      leer_otros_campos(personaaux)
      personaaux.ocupado ← '*'
      escribir(f, posi, personaaux)
   si_no
      escribir('No está')
   fin_si
   cerrar(f)
fin
```

9.10.2. Clave-dirección

Con respecto a las transformaciones clave-dirección deberemos realizar aún algunas consideraciones.

En un soporte direccionable —normalmente un disco—, cada posición se localiza por su dirección absoluta —número de pista y número de sector en el disco—. Los archivos directos manipulan direcciones relativas en lugar de absolutas, lo que hará al programa independiente de la posición absoluta del archivo en el soporte. Los algoritmos de conversión de clave transformarán las claves en direcciones relativas. Suponiendo que existen `N` posiciones disponibles para el archivo, los algoritmos de conversión de clave producirán una dirección relativa en el rango 1 a `N` por cada valor de la clave.

Existen varias técnicas para obtener direcciones relativas. En el caso en que dos registros distintos produzcan la misma dirección, se dice que se produce una colisión o sinónimo.

9.10.3. Tratamiento de las colisiones

Las colisiones son inevitables y, como se ha comentado, se originan cuando dos registros de claves diferentes producen la misma dirección relativa. En estos casos las colisiones se pueden tratar de dos formas diferentes.

Supongamos que un registro produce una dirección d1 que ya está ocupada. ¿Dónde colocar el nuevo registro? Existen dos métodos básicos:

- Considerar una zona de excedentes y asignar el registro a la primera posición libre en dicha zona. Fue el método aplicado en los algoritmos anteriores.
- Buscar una nueva dirección libre en la zona de datos del archivo.

9.10.4. Acceso a los archivos directos mediante indexación

La indexación es una técnica para el acceso a los registros de un archivo. En esta técnica el archivo principal de registros está suplementado por uno o más índices. Los índices pueden ser archivos independientes o un array que se carga al comenzar en la memoria del ordenador, en ambos casos estarán formados por registros con los campos código o clave y posición o número de registro.

El almacenamiento de los índices en memoria permite encontrar los registros más rápidamente que cuando se trabaja en disco.

Cuando se utiliza un archivo indexado se localizan los registros en el índice a través del campo clave y éste retorna la posición del registro en el archivo principal, directo.

Las operaciones básicas a realizar con un archivo indexado son:

- Crear las zonas de índice y datos como archivos vacíos originales.
- Cargar el archivo índice en memoria antes de utilizarlo.
- Reescribir el archivo índice desde memoria después de utilizarlo.
- Añadir registros al archivo de datos y al índice.
- Borrar registros.
- Actualizar registros en el archivo de datos.

Krasner Podres 1095584-2202180143

Altas

El procedimiento utilizado para dar las altas en el archivo anterior podría ser el siguiente:

```
procedimiento altas(E/S arr: a  E/S entero: n)
   var
   reg          : persona
   entero       : p
   lógico       : encontrado
   entero       : núm
inicio
   si n = max entonces
      escribir('lleno')
   si_no
      leer_reg(persona)
      encontrado ← falso
      búsqueda_binaria(a, n, persona.cod, p, encontrado)
      si encontrado entonces
         escribir('Clave duplicada')
      si_no
         núm ← LDA(f)/tamaño_de(datos_personales) + 1
         {Insertamos un nuevo registro en la tabla
            sin que pierda su ordenación }
         alta_índice(a, n, p, persona.cod, núm)
         n ← n + 1
         {Escribimos el nuevo registro al final del
            archivo principal }
         escribir(f, núm, persona)
      fin_si
   fin_si
   { en el programa principal, al terminar, crearemos de
   nuevo el archivo índice a partir de los registros
   almacenados en el array a }
fin_procedimiento
```

9.11. Procesamiento de archivos secuenciales indexados

Los archivos de organización secuencial indexada contienen tres áreas: una de datos que agrupa a los registros, otra de índice que contiene los niveles de índice y una zona de desbordamiento o excedentes para el caso de actualizaciones con adición de nuevos registros.

Los registros han de ser grabados obligatoriamente en orden secuencial ascendente por el contenido del campo clave y, simultáneamente a la grabación de los registros, el sistema crea los índices.

Una consideración adicional con respecto a este tipo de organización es que es posible usar más de una clave, hablaríamos así de la clave primaria y de una o más secundarias. El valor de la clave primaria es la base para la posición física de los registros en el archivo y debe ser única. Las claves secundarias pueden ser o no únicas y no afectan al orden físico de los registros.

9.12. Tipos de archivos: consideraciones prácticas en C/C++ y Java

Los archivos se pueden clasificar en función de determinadas características. Entre ellas las más usuales son: por el tipo de acceso o por la estructura de la información del archivo.

Krasner Podres 1095584-2202180146

Dirección del flujo de datos

Los archivos se clasifican en función del flujo de los datos o por el modo de acceso a los datos. En función de la dirección del flujo de los datos son de:

- *Entrada.* Aquellos cuyos datos se leen por parte del programa (archivos de lectura).
- *Salida.* Archivos que escribe el programa (archivos de escritura).
- *Entrada/Salida.* Archivos en los que se puede leer y escribir.

La determinación del tipo de archivo se realiza en el momento de la creación del archivo.

Tipos de acceso

Los archivos se clasifican en:

- *Secuenciales.* El orden de acceso a los datos es secuencial; primero se accede al primer elemento, luego al segundo y así sucesivamente.
- *Directos (aleatorios).* El acceso a un elemento concreto del archivo es directo. Son similares a las tablas.

Estructura de la información

Los archivos guardan información en formato binario y se distribuye en una secuencia o flujo de bytes. Teniendo en cuenta la información almacenada los archivos se clasifican en:

- ***Texto.*** En estos archivos se guardan solamente ciertos caracteres imprimibles, como letras, números y signos de puntuación, saltos de línea, etcétera. Están permitidos ciertos rangos de valores para cada *byte*. En un archivo de texto no está permitido el *byte* de final de archivo y si existe no se puede ver más allá de la posición donde está el *byte*.
- ***Binarios.*** Contienen cualquier valor que se pueda almacenar en un *byte*.

El tipo de información almacenada en los archivos se define a la hora de abrirlos (para lectura, escritura). Con posterioridad, cada operación leerá los bytes correspondientes al tipo de datos.

> Un archivo de texto es un caso particular de archivo de organización secuencial y es una serie continua de caracteres que se pueden leer uno tras otro. Cada registro de un archivo de texto es del tipo de cadena de caracteres.

El tratamiento de archivos de texto es elemental y en el caso de lenguajes como Pascal es posible detectar lecturas de caracteres especiales como final de archivo o final de línea.

9.12.1. Archivos de texto

Los archivos de texto también se denominan archivos ASCII y son legibles por los usuarios o programadores. Los terminales, los teclados y las impresoras tratan con datos carácter; así, cuando se desea escribir un número como "1234" en la pantalla se debe convertir a cuatro caracteres ("1", "2", "3", "4") y ser escritos en el terminal.

De modo similar cuando se lee un número de teclado, los datos se deben convertir de caracteres a enteros. En el caso del lenguaje C++ esta operación se realiza con el operador >>. Las computadoras trabajan con datos binarios. Cuando se leen números de un archivo ASCII, el programa debe procesar los datos carácter a través de una rutina de conversión, lo que entraña grandes recursos. Los archivos binarios, por el contrario, no requieren conversión e incluso ocupan menos espacio que los archivos ASCII; su gran inconveniente es que los archivos binarios no se pueden imprimir directamente en una impresora ni visualizar en un terminal.

Krasner Podres 1095584-2202180147

Los archivos ASCII son *portables* (en la mayoría de los casos) y se pueden mover de una computadora a otra sin grandes problemas. Sin embargo, los archivos binarios son prácticamente no portables; a menos que sea un programador experto es casi imposible hacer portable un archivo binario.

9.12.2. Archivos binarios

Los archivos binarios contienen cualquier valor que se puede almacenar en un *byte*. En estos archivos su final no se almacena como un *byte* concreto, se escriben copiando una imagen del contenido de un segmento de la memoria al disco y por consiguiente los valores numéricos aparecen como unos caracteres extraños que se corresponden con la codificación de dichos valores en la memoria de la computadora, aunque en apariencia son prácticamente indescifrables para el programador o el usuario.

Cuando se intenta abrir un archivo binario con el editor aparecerán secuencias de caracteres como:

```
E # @ % Â a ^ ^ ...
```

¿Cuál es el archivo más recomendable para utilizar? En la mayoría de los casos, el ASCII es el mejor. Si se tienen pequeñas a medianas cantidades de datos, el tiempo de conversión no afecta seriamente a su programa. Igualmente los archivos ASCII facilitan la verificación de los datos. Por el contrario, sólo cuando se utilizan grandes cantidades de datos los problemas de espacio y rendimiento, aconsejarán utilizar formatos binarios.

Los archivos de texto suelen denominarse con la extensión `.txt`, mientras que los archivos binarios suelen tener la extensión `.dat`. Los primeros son muy eficientes para intercambiar datos entre aplicaciones y para proporcionar datos de entrada de programas que se deban ejecutar varias veces; por el contrario, son poco eficientes para manejar grandes volúmenes de información o bases de datos. Por otra parte, todos los archivos binarios permiten acceso directo, lo cual es muy útil para manejar grandes archivos o bases de datos, ya que se puede ir directamente a leer el registro n sin tener que leer antes el primero, el segundo,..., el n – 1 registros anteriores

Krasner Podres 1095584-2202180148

9.12.3. Lectura y escritura de archivos

Las operaciones típicas sobre un archivo son: creación, lectura y escritura. En el caso de C++ los archivos se manipulan mediante un tipo de objeto flujo. Normalmente los objetos que se usan para tratar con archivos se denominan **archivos lógicos**. Los **archivos físicos** son aquellos que almacenan realmente la información en disco (o dispositivos de memoria secundaria correspondiente: disco duro, discos ópticos, memorias *flash*, etcétera).

En el caso del lenguaje C++, para todas las operaciones con archivo se requiere utilizar la biblioteca de cabecera fstream.h por lo que es preciso que los programas inserten la sentencia:

```
#include <fstream.h>
```

o bien, en el caso de ANSI C++ estándar:

```
#include <fstream>
using namespace std;
```

En general, todo tratamiento de un archivo consta de tres pasos importantes:

- *Apertura del archivo.* El modo de implementar la operación dependerá de si un archivo es de lectura o escritura.
- *Acceso al archivo.* En esta etapa se llega o imprimen los datos.
- *Cierre del archivo.* Actualiza el archivo y se elimina la información no significativa.

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

9.1. *Escribir un algoritmo que permita la creación e introducción de los primeros datos en un archivo secuencial,* PERSONAL, *que deseamos almacene la información mediante registros de siguiente tipo.*

```
tipo
   registro:     datos_personales
      <tipo_dato1>      :      nombre_campo1
      <tipo_dato2>      :      nombre_campo2
      ............      :      ............
fin_registro
```

Análisis del problema

Tras la creación y apertura en modo conveniente del archivo, el algoritmo solicitará la introducción de datos por teclado y los almacenará de forma consecutiva en el archivo.

Se utilizará una función, `último_dato(persona)`, para determinar el fin en la introducción de datos.

Diseño del algoritmo

```
algoritmo   Ejercicio_9_1
tipo
   registro:     datos_personales
      <tipo_dato1>      :      nombre_campo1
      <tipo_dato2>      :      nombre_campo2
      ............      :      ............
   fin_registro
   archivo_s de  datos_personales: arch
var
   arch             :  f
   datos_personales :  persona
inicio
   crear (f, 'Personal')
   abrir (f, e, 'Personal')
   llamar_a leer_reg (persona)
   //     Procedimiento para la lectura de un
   //     registro campo a campo
   mientras no último_dato(persona) hacer
      llamar_a escribir_f_reg (f,   persona)
      // Procedimiento auxiliar, no desarrollado, para la
      // escritura en el archivo del registro campo a campo
      llamar_a leer_reg(persona)
   fin_mientras
   cerrar (f)
fin
```

9.2. *Supuesto que deseamos añadir nueva información al archivo* PERSONAL, *anteriormente creado, diseñar el algoritmo correspondiente.*

Análisis del problema

Al abrir el archivo para escritura se coloca el puntero de datos al final de éste, permitiéndonos, con un algoritmo similar al anterior, la adición de nueva información al finalizarlo.

Krasner Podres 1095584-2202180158

Diseño del algoritmo

```
algoritmo Ejercicio_9_2
tipo
   registro: datos_personales
      <tipo_dato1> : nombre_campo1
      <tipo_dato2> : nombre_campo2
      ............ : ............
   fin_registro
   archivo_s de  datos_personales: arch
var
   arch             : f
   datos_personales : persona
inicio
   abrir (f, e, 'PERSONAL')
   llamar_a leer_reg (persona)
   mientras no último_dato (persona) hacer
      llamar_a escribir_f_reg (f, persona)
      llamar_a leer_reg (persona)
   fin_mientras
   cerrar (f)
fin
```

9.3. *Diseñar un algoritmo que muestre por pantalla el contenido de todos los registros del archivo* PERSONAL.

Análisis del problema

Se debe abrir el archivo para lectura y, repetitivamente, leer los registros y mostrarlos por pantalla hasta detectar el fin de fichero.

Se considera que la función FDA(id_arch) detecta el final de archivo con la lectura de su último registro.

Diseño del algoritmo

```
algoritmo Ejercicio_9_3
tipo
   registro: datos_personales
      <tipo_dato1>: nombre_campo1
      <tipo_dato2>: nombre_campo2
      ............: ............
   fin_registro
   archivo_s de datos_personales: arch
var
   arch             : f
   datos_personales : persona
inicio
   abrir (f, l, 'PERSONAL')
   mientras no fda (f) hacer
      llamar_a leer_f_reg (f, persona)
      llamar_a escribir_reg (persona)
   fin_mientras
   cerrar (f)
fin
```

Si se considera la existencia de un registro especial que marca el fin de archivo, la función FDA(id_arch) se activaría al leer este registro y es necesario modificar un poco nuestro algoritmo.

Krasher Podres 1095554-2202180159

```
inicio
   abrir (f,1, 'PERSONAL')
   llamar_a leer_f_reg (f, persona)
   mientras no fda (f) hacer
      llamar_a escribir_reg (persona)
      llamar_a leer_f_reg (f, persona)
   fin_mientras
   cerrar (f)
fin
```

9.4. *Una librería almacena en un archivo secuencial la siguiente información sobre cada uno de sus libros: CÓDIGO, TÍTULO, AUTOR y PRECIO.*

El archivo está ordenado ascendentemente por los códigos de los libros –de tipo cadena–, que no pueden repetirse.

Se precisa un algoritmo con las opciones:

1. *Insertar:* *Permitirá insertar nuevos registros en el archivo, que debe mantenerse ordenado en todo momento.*
2. *Consulta.* *Buscará registros por el campo CÓDIGO.*

Análisis del problema

El algoritmo comenzará presentando un menú de opciones a través del cual se haga posible la selección de un procedimiento u otro.

`Insertar:` Para poder colocar el nuevo registro en el lugar adecuado, sin que se pierda la ordenación inicial, se necesita utilizar un archivo auxiliar. En dicho archivo se van copiando los registros hasta llegar al punto donde debe colocarse el nuevo, entonces se escribe y continúa con la copia de los restantes registros.

`Consulta:` Como el archivo está ordenado y los códigos no repetidos, el proceso de consulta se puede acelerar. Se recorre el archivo de forma secuencial hasta encontrar el código buscado, o hasta que éste sea menor que el código del registro que se acaba de leer desde el archivo, o bien, si nada de esto ocurre, hasta el fin del archivo.

Cuando el código buscado es menor que el código del registro que se acaba de leer desde el archivo, se puede deducir que de ahí en adelante ese registro ya no podrá estar en el fichero, por tanto, se puede abandonar la búsqueda.

Diseño del algoritmo

```
algoritmo Ejercicio_9_4
tipo
   registro  : reg
      cadena : cod
      cadena : título
      cadena : autor
      entero : precio
   fin_registro
   archivo_s de reg : arch
var
   entero : op

inicio
   repetir
      escribir( 'MENÚ')
      escribir( '1.- INSERTAR')
      escribir( '2.- CONSULTA')
      escribir( '3.- FIN')
```

Krasner Podres 1095564-2202180202

```
      escribir('Elija opción')
      leer (op)
      según_sea op hacer
         1 : llamar_a insertar
         2 : llamar_a consulta
      fin_según
   hasta_que op = 3
fin

procedimiento insertar
var
   arch     : f, f2
   reg      : rf, r
   lógico   : escrito
   carácter : resp
inicio
   repetir
      abrir (f,1,'Libros.dat')
      crear (f2, 'Nlibros.dat')
      abrir (f2,e, 'Nlibros.dat')
      escribir ('Deme el código')
      leer (r.cod)
      escrito ← falso
      mientras no FDA(f) hacer
         llamar_a leer_arch_reg ( f, rf)
         si rf.cod > r.cod y no escrito entonces
         // si se lee del archivo un registro con código
         // mayor que el nuevo y éste aún no se
         // ha escrito, es el momento de insertarlo
            escribir('Deme otros campos')
            llamar_a completar ( r )
            llamar_a escribir_arch_reg ( f2, r )
            escrito ← verdad
            // Se debe marcar que se ha escrito
            // para que no siga insertándose, desde aquí
            // en adelante
         si_no
            si rf.cod = r.cod entonces
               escrito ← verdad
            fin_si
         fin_si
         llamar_a escribir_arch_reg ( f2, rf )
         // De todas formas se escribe el que
         // se lee del archivo
      fin_mientras
      si no escrito entonces
      // Si el código del nuevo es mayor que todos los del
      // archivo inicial, se llega al final sin haberlo
      // escrito
         escribir ('Deme otros campos')
         llamar_a completar (r)
         llamar_a escribir_arch_reg ( f2, r )
      fin_si
         cerrar (f, f2)
```

Krasner Padres 1095564-2202181413
1095564220218141 3

```
   borrar ('Libros.dat')
   renombrar ('Libros.dat', 'Libros.dat')
   escribir ('¿Seguir? (s/n)')
   leer ( resp )
   hasta_que resp = 'n'
fin_procedimiento

procedimiento consulta
var
   reg: rf, r
   arch: f
   carácter: resp
   lógico: encontrado, pasado
inicio
   resp ← 's'
   mientras resp <> 'n' hacer
      abrir (f, 1, 'Libros.dat')
      escribir ('Deme el código a buscar ')
      leer ( r.cod)
      encontrado ← falso
      pasado ← falso
      mientras no FDA (f) y no encontrado y no pasado hacer
         llamar_a leer_arch_reg (f, rf)
         si r.cod = rf.cod entonces
            encontrado ← verdad
            llamar_a escribir_reg ( rf )
         si_no
            si r.cod < rf.cod entonces
               pasado ← verdad
            fin_si
         fin_si
      fin_mientras
      si no encontrado entonces
         escribir ( 'Ese libro no está')
      fin_si
      cerrar (f)
      escribir ('¿Seguir? (s/n)')
      leer ( resp )
   fin_mientras
fin_procedimiento
```

9.5. *Diseñar un algoritmo que efectúe la creación de un archivo directo —*PERSONAL*—, cuyos registros serán del siguiente tipo:*

```
tipo
registro: datos_personales
   <tipo_dato1> : cod   // Campo clave
   ............  : ..............
   <tipo_datoN> : nombre_campoN
fin_registro
```

y en el que, posteriormente, vamos a introducir la información que utiliza el método de transformación de clave.

Krasner Podres 1095584-2202181413

Análisis del problema

El método de transformación de claves consiste en introducir los registros, en el soporte que los va a contener, en la dirección que proporciona el algoritmo de conversión. Su utilización obliga al almacenamiento del código en el propio registro y hace conveniente la inclusión en el registro de un campo auxiliar —ocupado— en el que se marque si el registro está o no ocupado. Durante el proceso de creación se debe realizar un recorrido de todo el archivo inicializando el campo ocupado a vacío, por ejemplo, a espacio.

Diseño del algoritmo

```
algoritmo Ejercicio_9_5
const
   Max = <valor>
tipo
   registro: datos_personales
      <tipo_dato1> : cod // Podría no ser necesario
                         // su almacenamiento, en el caso
                         // de que coincidiera con el
                         // índice
      ............ : ............
      <tipo_daton> : nombre_campon
   fin_registro
   archivo_d de datos_personales: arch
var
   arch             : f
   datos personales : persona
   entero           : i
inicio
   crear (f, 'PERSONAL')
   abrir (f,1/e, 'PERSONAL')
   desde i ← 1 hasta Max hacer
      persona.ocupado
      escribir (f, persona, i)
   fin_desde
   cerrar (f)
fin
```

9.6. *Se desea introducir información, por el método de transformación de clave, en el archivo PERSONAL creado en el ejercicio anterior, diseñar el algoritmo correspondiente.*

Análisis del problema

Como anteriormente se ha explicado, el método de transformación de claves consiste en introducir los registros en el soporte que los va a contener, en la dirección que proporciona el algoritmo de conversión.

A veces, registros distintos, sometidos al algoritmo de conversión, proporcionan una misma dirección, por lo que se debe tener previsto un espacio en el disco para el almacenamiento de los registros que han consolidado. Aunque se puede hacer de diferentes maneras, en este caso se reserva espacio para las colisiones en el propio fichero a continuación de la zona de datos.

Se supone que la dirección más alta capaz de proporcionar el algoritmo de conversión es `Findatos` y se colocan las colisiones que se produzcan a partir de allí en posiciones consecutivas del archivo.

La inicialización a espacio del campo ocupado se realiza hasta `Max`, dando por supuesto que `Max` es mayor que `Fin-datos`.

Krasher Padres 1095564-220218413

Diseño del algoritmo

```
algoritmo Ejercicio_9_6
const
   Findatos = <valor1>
   Max      = <valor2>
tipo
   registro: datos_peronales
      <tipo_dato1> : cod // Podría no ser necesario
                         // su almacenamiento, en el caso
                         // de que coincidiera con el
                         // índice
      ............ : ............
      <tipo_daton> : nombre_campon
   fin_registro
   archivo_d de datos_personales: arch
var
   arch             : f
   datos personales : persona, personaaux
   lógico           : encontradohueco
   entero           : i
inicio
   abrir (f,1/e, 'PERSONAL')
   leer (personaaux.cod)
   posi ← HASH (personaaux.cod)
   // HASH es el nombre de la función de transformación de
   // claves. La cual devolverá valores
   // entre 1 y Findatos, ambos inclusive
leer(f, persona, posi)
si persona.ocupado = '*' entonces //El '*' indica que está
                                  //ocupado
   encontradohueco ← falso
   posi ← Findatos
   mientras posi < Max y no encontradohueco hacer
      posi ← posi + 1
      leer(f, persona, posi)
      si persona.ocupado <> '*' entonces
            encontradohueco ← verdad
         fin_si
      fin_mientras
   si_no
      encontradohueco ← verdad
   fin_si
   si encontradohueco entonces
      llamar_a leer_otros_campos (personaaux)
      persona ← personaaux
      persona.ocupado ← '*' //Al dar un alta marcaremos
                            //el campo ocupado
      escribir(f, persona, posi)
   si_no
      escribir ('No está')
   fin_si
   cerrar (f)
fin
```

Krasher Podres 1095564-2202161414

1095564220216141 4

Conceptos clave

- Archivos de texto
- Concepto de flujo
- Organización de archivos
- Organización directa
- Organización secuencial
- Organización secuencial indexada
- Registro físico
- Registro lógico

Resumen

Un archivo de datos es un conjunto de datos relacionados entre sí y almacenados en un dispositivo de almacenamiento externo. Estos datos se encuentran estructurados en una colección de entidades denominadas artículos o registros, de igual tipo, y que constan a su vez de diferentes entidades de nivel más bajo denominadas campos. Un archivo de texto es el que está formado por líneas, constituidas a su vez por una serie de caracteres, que podrían representar los registros en este tipo de archivos. Por otra parte, los archivos pueden ser binarios y almacenar no sólo caracteres sino cualquier tipo de información tal y como se encuentra en memoria.

1. Java y C# realizan las operaciones en archivos a través de flujos, manipulados por clases, que conectan con el medio de almacenamiento. De forma que para crear, leer o escribir un archivo se requiere utilizar una clase que defina la funcionalidad del flujo. Los flujos determinan el sentido de la comunicación (lectura, escritura o lectura/escritura), el posicionamiento directo o no en un determinado registro y la forma de leer y/o escribir en el archivo. Pueden utilizarse flujos de bytes cadenas o tipos primitivos. La personalización de flujos se consigue por asociación o encadenamiento de otros flujos con los flujos base de apertura de archivos.
2. Registro lógico es una colección de información relativa a una entidad particular. El concepto de registro es similar al de estructura desde el punto de vista de que permiten almacenar datos de tipo heterogéneo.
3. Registro físico es la cantidad más pequeña de datos que pueden transferirse en una operación de entrada/salida entre la memoria central y los dispositivos.
4. La organización de archivos define la forma en la que los archivos se disponen sobre el soporte de almacenamiento y puede ser secuencial, directa o secuencial-indexada.
5. La organización secuencial implica que los registros se almacenan unos al lado de otros en el orden en el que van siendo introducidos y que para efectuar el acceso a un determinado registro es necesario pasar por los que le preceden.
6. Los archivos de texto se consideran una clase especial de archivos secuenciales.
7. En la organización directa el orden físico de los registros puede no corresponderse con aquel en el que han sido introducidos y el acceso a un determinado registro no obliga a pasar por los que le preceden. Para poder acceder a un determinado registro de esta forma se necesita un soporte direccionable y la longitud de los registros debe ser fija.
8. La organización secuencial-indexada requiere la existencia de un área de datos, un área de índices, un área de desbordamiento o colisiones y soporte direccionable.

Krasner Padres 1995564-2202181415

EJERCICIOS

9.1. Diseñar un algoritmo que permita crear un archivo AGENDA de direcciones cuyos registros constan de los siguientes campos:

NOMBRE
DIRECCIÓN
CIUDAD
CÓDIGO POSTAL
TELÉFONO
EDAD

9.2. Realizar un algoritmo que lea el archivo AGENDA e imprima todos los registros.

9.3. Diseñar un algoritmo que copie el archivo secuencial AGENDA de los ejercicios anteriores en un archivo directo DIRECTO_AGENDA, de modo que cada registro mantenga su posición relativa.

9.4. Se dispone de un archivo indexado denominado DIRECTORIO, que contiene los datos de un conjunto de personas y cuya clave es el número del DNI. Escribir un algoritmo capaz de realizar una consulta de un registro. Si no se encuentra el registro se emite el correspondiente mensaje de ERROR.

9.5. Se dispone de un archivo STOCK correspondiente a la existencia de artículos de un almacén y se desea señalar aquellos artículos cuyo nivel está por debajo del mínimo y que visualicen un mensaje "hacer pedido". Cada artículo contiene un registro con los siguientes campos: número de código del artículo, nivel mínimo, nivel actual, proveedor, precio.

9.6. El director de un colegio desea realizar un programa que procese un archivo de registros correspondiente a los diferentes alumnos del centro, a fin de obtener los siguientes datos:

- Nota más alta y número de identificación del alumno correspondiente.
- Nota media por curso.
- Nota media del colegio.

NOTA: *Si existen varios alumnos con la misma nota más alta, se deberán visualizar todos ellos.*

9.7. Diseñar un algoritmo que genere un archivo secuencial BIBLIOTECA, cuyos registros contienen los siguientes campos:

TÍTULO
AUTOR
EDITORIAL
AÑO DE EDICIÓN
ISBN
NÚMERO DE PÁGINAS

9.8. Diseñar un algoritmo que permita modificar el contenido de alguno de los registros del archivo secuencial BIBLIOTECA mediante datos introducidos por teclado.

Krasher Padres 1095584-2202181416

CAPÍTULO 10

Ordenación, búsqueda e intercalación

10.1. Introducción
10.2. Ordenación
10.3. Búsqueda
10.4. Intercalación

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Krasner Podnes 1095564-2202161417

INTRODUCCIÓN

1095564-2202161417

Las computadoras utilizan la mayoría de su tiempo en operaciones de *búsqueda*, *clasificación* y *mezcla de datos*. Las operaciones de cálculo numérico, y sobre todo de gestión, requieren normalmente operaciones de clasificación de los datos: ordenar fichas de clientes por orden alfabético, por direcciones o por código postal. Existen dos métodos de ordenación: *interna* (de ***arrays***, **arreglos**) y *externa* (archivos). Los arrays se almacenan en la memoria interna o central, de acceso aleatorio y directo, y por ello su gestión es rápida. Los *archivos* se sitúan adecuadamente en dispositivos de almacenamiento externo que son más lentos y basados en dispositivos mecánicos: cintas y discos magnéticos o bien en la nube. Las técnicas de ordenación, búsqueda y mezcla son muy importantes y el lector deberá dedicar especial atención al conocimiento y aprendizaje de los diferentes métodos que en este capítulo se analizan.

10.1. Introducción

Ordenación, búsqueda y, en menor medida, *intercalación*, son operaciones básicas en el campo de la documentación y en las que, según señalan las estadísticas, las computadoras emplean la mitad de su tiempo.

Aunque su uso puede ser con vectores (arrays) y con archivos, este capítulo se referirá a vectores.

La *ordenación (clasificación)* es la operación de organizar un conjunto de datos en algún orden que puede ser creciente o decreciente en datos numéricos, o bien en orden alfabético directo o inverso. Operaciones típicas de ordenación son: lista de números, archivos de clientes de banco, nombres de una agenda telefónica, etcétera. En síntesis, la ordenación significa poner objetos en orden (numérico para los números y alfabético para los caracteres) ascendente o descendente.

Por ejemplo, las clasificaciones de los equipos de fútbol de la liga en la 1a. división española se pueden organizar alfabéticamente de manera creciente/decreciente, o bien por clasificación numérica ascendente/descendente.

Los nombres de los equipos y los puntos de cada uno se almacenan en dos vectores:

```
equipo [1] = 'Real Madrid'        puntos [1] = 10
equipo [2] = 'Barcelona'          puntos [2] = 14
equipo [3] = 'Valencia'           puntos [3] =  8
equipo [4] = 'Oviedo'             puntos [4] = 12
equipo [5] = 'Betis'              puntos [5] = 16
```

Si los vectores se ponen en orden decreciente de puntos de clasificación:

```
equipo [5] = 'Betis'              puntos [5] = 16
equipo [2] = 'Barcelona'          puntos [2] = 14
equipo [4] = 'Oviedo'             puntos [4] = 12
equipo [1] = 'Real Madrid'        puntos [1] = 10
equipo [3] = 'Valencia'           puntos [3] =  8
```

Los nombres de los equipos y los puntos conseguidos en el campeonato de liga anterior, ordenados de modo alfabético serían:

```
equipo [1] = 'Barcelona'          puntos [1] =  5
equipo [2] = 'Cádiz'              puntos [2] = 13
equipo [3] = 'Málaga'             puntos [3] = 12
equipo [4] = 'Oviedo'             puntos [4] =  8
equipo [5] = 'Real Madrid'        puntos [5] =  4
equipo [6] = 'Valencia'           puntos [6] = 16
```

o bien se pueden situar en orden numérico decreciente:

```
equipo [6] = 'Valencia'           puntos [6] = 16
equipo [2] = 'Cádiz'              puntos [2] = 13
equipo [3] = 'Málaga'             puntos [3] = 12
equipo [4] = 'Oviedo'             puntos [4] =  8
equipo [1] = 'Barcelona'          puntos [1] =  5
equipo [5] = 'Real Madrid'        puntos [5] =  4
```

Los vectores anteriores comienzan en orden alfabético de equipos y se reorganizan de forma descendente de acuerdo con los "puntos". La lista telefónica se clasifica en orden alfabético de abonados; un archivo de clientes de una entidad bancaria normalmente se clasifica en orden ascendente de números de cuenta. El propósito final de la clasificación es facilitar la manipulación de datos en un vector o en un archivo.

Algunos autores diferencian entre un conjunto o *vector clasificado (sorted)* y *vector ordenado (ordered set)*. Un *conjunto ordenado* es aquel en el que el orden de aparición de los elementos afecta al significado de la estructura

Krasher Padres 1995564-2202181418

completa de datos: puede estar clasificado, pero no es imprescindible. Un *conjunto clasificado* es aquel en que los valores de los elementos han sido utilizados para disponerlos en un orden particular: es, probablemente, un conjunto ordenado, pero no necesariamente.

Es importante estudiar la clasificación por dos razones. Una es que la clasificación de datos es tan frecuente que todos los usuarios de computadoras deben conocer estas técnicas. La segunda es que, si bien es una aplicación que se puede describir fácilmente, es bastante difícil conseguir el diseño y escritura de buenos algoritmos.

La clasificación de los elementos numéricos del vector:

```
7, 3, 2, 1, 9, 6, 7, 5, 4
```

en orden ascendente producirá:

```
1, 2, 3, 4, 5, 6, 7, 7, 9
```

Obsérvese que pueden existir elementos de igual valor dentro de un vector.

Existen muchos algoritmos de clasificación, con diferentes ventajas e inconvenientes. Uno de los objetivos de éste y del capítulo 11 es el estudio de los métodos de clasificación más usuales y de mayor aplicación.

La *búsqueda* de información es, al igual que la ordenación, otra operación muy frecuente en el tratamiento de información; es una actividad que se realiza diariamente en cualquier aspecto de la vida: búsqueda de palabras en un diccionario, nombres en una guía telefónica, localización de libros en una librería, entre otros. A medida que la información se almacena en una computadora, su recuperación y búsqueda se convierte en una tarea principal de dicha computadora.

Krasher Padres 1095564-2202161419

10.2. Ordenación

En un vector es necesario, con frecuencia, clasificar sus elementos en un orden particular; por ejemplo, clasificar un conjunto de números en orden creciente o una lista de nombres por orden alfabético. La clasificación es una operación tan frecuente en programas de computadora que una gran cantidad de algoritmos se han diseñado para clasificar listas de elementos con eficacia y rapidez. La elección de un determinado algoritmo depende del tamaño del vector o **array** (**arreglo**) a clasificar, el tipo de datos y la cantidad de memoria disponible.

La *ordenación* o *clasificación* es el proceso de organizar datos en algún orden o secuencia específica, tal como creciente o decreciente para datos numéricos o alfabéticamente para datos de caracteres. Los *métodos de ordenación* se dividen en dos categorías:

- **Ordenación de vectores, tablas (arrays o arreglos).**
- **Ordenación de archivos.**

La ordenación de arrays se denomina también *ordenación interna*, ya que se almacena en la memoria interna de la computadora de gran velocidad y acceso aleatorio. La ordenación de archivos se suele hacer casi siempre sobre soportes de almacenamiento externo, discos, cintas, etcétera, y, por ello, se denomina también *ordenación externa*. Estos dispositivos son más lentos en las operaciones de entrada/salida, pero, por el contrario, pueden contener mayor cantidad de información.

Ordenación interna: clasificación de los valores de un vector según un orden en memoria central: rápida.
Ordenación externa: clasificación de los registros de un archivo situado en un soporte externo: menos rápido.

EJEMPLO

Clasificación en orden ascendente del vector:

```
7, 3, 2, 1, 9, 6, 7, 5, 4
```

se obtendrá el nuevo vector:

```
1, 2, 3, 4, 5, 6, 7, 7, 9
```

Los métodos de clasificación se explicarán aplicados a vectores (arrays unidimensionales), pero se pueden extender a matrices o tablas (arrays o arreglos bidimensionales), considerando la ordenación respecto a una fila o columna.

Los *métodos directos* son los que se realizan en el espacio ocupado por el array. Los más populares son:

- *Intercambio*.
- *Selección*.
- *Inserción*.

10.2.1. Método de intercambio o de burbuja

El algoritmo de clasificación de *intercambio* o de *burbuja* se basa en el principio de comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados.

Supongamos que se desea clasificar en orden ascendente el vector o lista

50	15	56	14	35	1	12	9
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Los pasos a seguir son:

1. Comparar A[1] y A[2]; si están en orden, se mantienen como están; en caso contrario, se intercambian entre sí.
2. A continuación, se comparan los elementos 2 y 3; de nuevo se intercambian si es necesario.
3. El proceso continúa hasta que cada elemento del *vector* ha sido comparado con sus elementos adyacentes y se han realizado los intercambios necesarios.

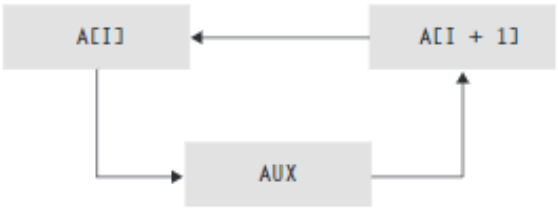
El método expresado en pseudocódigo en el primer diseño es:

```
desde I ← 1 hasta 7 hacer
   si elemento[I] > elemento[I + 1] entonces
      intercambiar (elemento[I], elemento [I + 1])
   fin_si
fin_desde
```

La acción *intercambiar* entre sí los valores de dos elementos A[I], A[I+1] es una acción compuesta que contiene las siguientes acciones, considerando una variable auxiliar AUX:

```
AUX ← A[I]
A[I] ← A[I+1]
A[I+1] ← AUXI
```

En realidad, el proceso gráfico es:



El elemento cuyo valor es mayor sube posición a posición hacia el final de la lista, al igual que las burbujas de aire en un depósito o botella de agua. Tras realizar un recorrido completo por todo el vector, el elemento mencionado habrá subido en la lista y ocupará la última posición. En el segundo recorrido, el segundo elemento llegará a la penúltima, y así sucesivamente.

En el ejercicio citado anteriormente los pasos sucesivos con cada una de las operaciones se muestran en las figuras 10.1 y 10.2.

Krasher Padres 1095564-2202181420

Vector inicial	1a. comp.	2a. comp.	...					
A[1]	50	15	15	15	15	15	15	15
A[2]	15	50	50	50	50	50	50	50
A[3]	56	56	56	14	14	14	14	14
A[4]	14	14	14	56	35	35	35	35
A[5]	35	35	35	35	56	1	1	1
A[6]	1	1	1	1	1	56	12	12
A[7]	12	12	12	12	12	12	56	9
A[8]	9	9	9	9	9	9	9	56

Figura 10.1. Método de la burbuja (paso 1).

Si se efectúa $n - 1$ veces la operación sobre una tabla de n valores se tiene ordenada la tabla. Cada operación requiere $n - 1$ comprobaciones o test y como máximo $n - 1$ intercambios. La ordenación total exigirá un máximo de $(n - 1) * (n - 1) = (n - 1)^2$ intercambios de elementos.

Los estados sucesivos del vector se indican en la figura 10.2:

Estado inicial	Después de paso 1	Después de paso 2
50	15	15
15	50	14
56	14	35
14	35	1
35	1	12
1	12	9
12	9	50
9	56	56

Figura 10.2. Método de la burbuja (paso 2).

Cristian Pilapanta 1095211-2202112327

EJEMPLO 10.1

Describir los diferentes pasos para clasificar en orden ascendente el vector:

72 64 50 23 84 18 37 99 45 8

Las operaciones sucesivas en cada uno de los pasos necesarios hasta obtener la clasificación final se muestran en la tabla 10.1.

Tabla 10.1. Pasos necesarios de la ordenación por burbuja.

Vector desordenado	Número de paso						Fin de clasificación		
	1	2	3	4	5	6	7	8	9
72	64	50	23	23	18	18	18	18	8
64	50	23	50	18	23	23	23	8	18
50	23	64	18	37	37	37	8	23	23
23	72	18	37	50	45	8	37	37	37
84	18	37	64	45	8	45	45	45	45
18	37	72	45	8	50	50	50	50	50
37	84	45	8	64	64	64	64	64	64
99	45	8	72	72	72	72	72	72	72
45	8	84	84	84	84	84	84	84	84
8	99	99	99	99	99	99	99	99	99

Método 1

El algoritmo se describirá, como siempre, con un diagrama de flujo y un pseudocódigo.

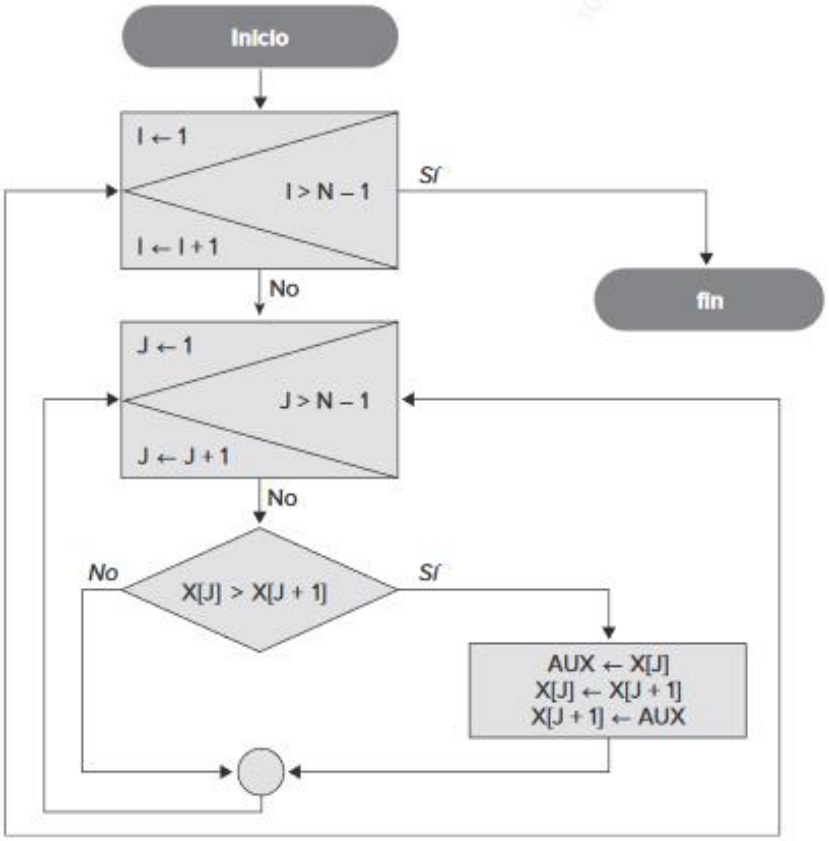
Pseudocódigo

```
algoritmo burbuja1
//incluir las declaraciones precisas//
inicio
   //lectura del vector//
   desde i ← 1 hasta N hacer
      leer(X[I])
   fin_desde
   //clasificación del vector
   desde I ←1 hasta N - 1 hacer
      desde J ←1 hasta J ← N - 1 hacer
         si X[j] > X[J+1] entonces
         //intercambiar
            AUX ← X[J]
            X[J] ←X[J+1]
            X[J+1] ←AUXI
         fin_si
      fin_desde
   fin_desde
   //imprimir lista clasificada
  desde J ←1 hasta N hacer
      escribir(X[J])
   fin_desde
fin
```

Diagrama de flujo 10.1

Para clasificar el vector completo se deben realizar las sustituciones correspondientes `(N - 1) * (N - 1)` o bien `N2 - 2N + 1` veces. Así, en el caso de un vector de cien elementos `(N = 100)` se deben realizar casi 10 000 iteraciones.

El algoritmo de clasificación es:



Cristian Pilaparita 1095211-2202112327

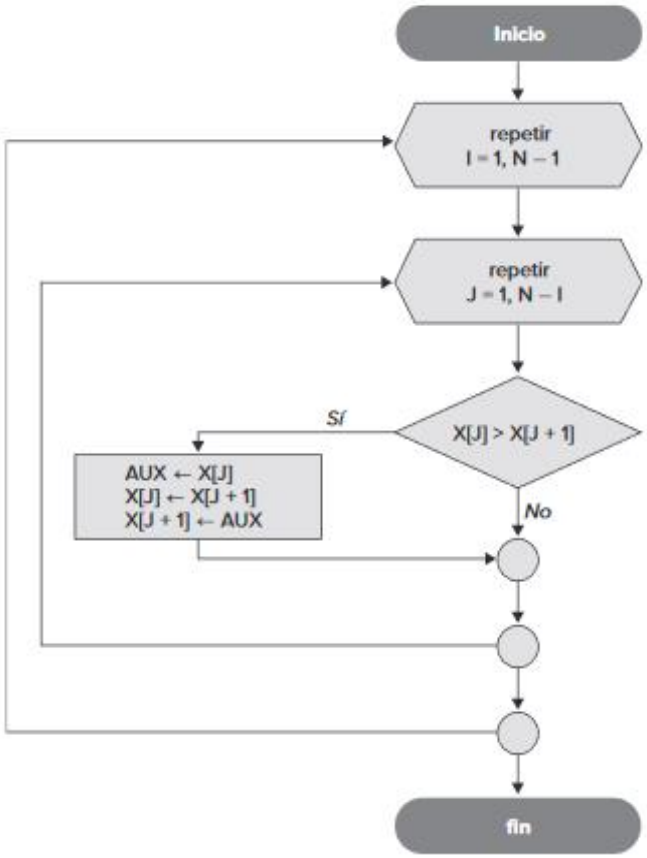
Método 2

Se puede realizar una *mejora en la velocidad de ejecución del algoritmo*. Observe que en el primer recorrido del vector (cuando I = 1) el valor mayor del vector se mueve al último elemento X[N]. Por consiguiente, en el siguiente paso no es necesario comparar X[N - 1] y X[N]. En otras palabras, el límite superior del bucle desde puede ser N - 2. Después de cada paso se puede decrementar en uno el límite superior del bucle desde. El algoritmo sería:

Pseudocódigo

```
algoritmo burbuja2
   //declaraciones
inicio
   //...
   desde I ← 1 hasta N - 1 hacer
      desde J ← 1 hasta N - I hacer
         si X[J] > X[J+1] entonces
            AUX ← X[J]
            X[J] ← X[J+1]
            X[J+1] ← AUX
         fin_si
      fin_desde
   fin_desde
fin
```

Diagrama de flujo 10.2

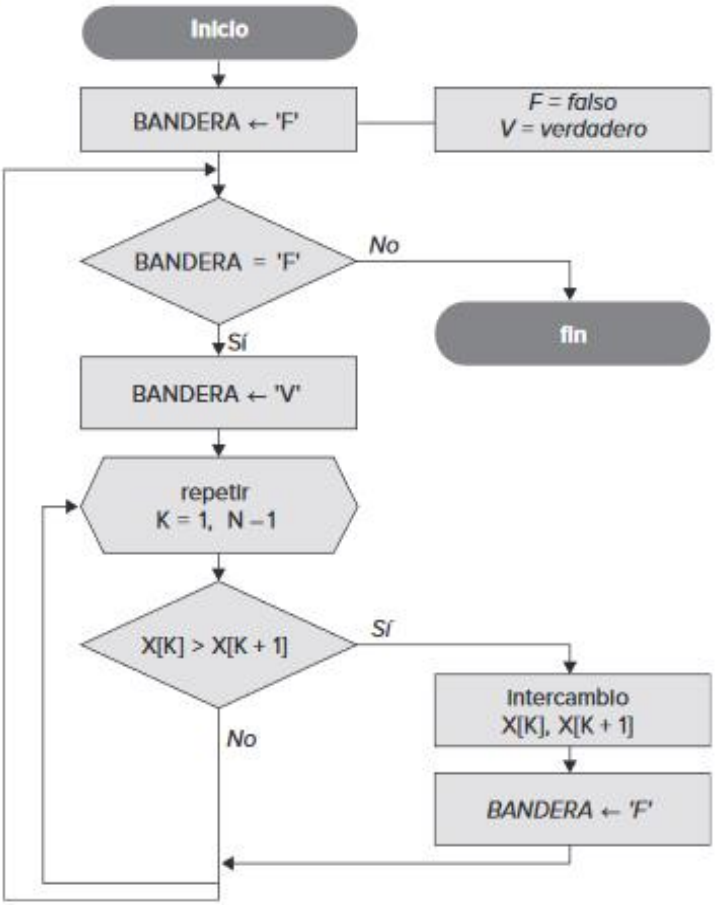


Cristian Pilapaña 1095211-2202112327

Método 3 (uso de una bandera/indicador)

Mediante una **bandera/indicador** o **centinela** (**switch**) o bien una variable lógica, se puede detectar la presencia o ausencia de una condición. Así, mediante la variable BANDERA se representa *clasificación terminada* con un valor verdadero y *clasificación no terminada* con un valor falso.

Diagrama de flujo 10.3



Pseudocódigo

```
algoritmo burbuja 3
   //declaraciones
inicio
   //lectura del vector
   BANDERA ← 'F'  //    F, falso; V, verdadero
i ← 1
   mientras (BANDERA = 'F') Y (i < N ) hacer
      BANDERA ← 'V'
      desde K ← 1 hasta N - i hacer
         si X[K] > X[K+1] entonces
            intercambiar(X[K],X[K + 1])
            //llamada a procedimiento intercambio
            BANDERA ← 'F'
         fin_si
      fin_desde
      i ← i+1
   fin_mientras
fin
```

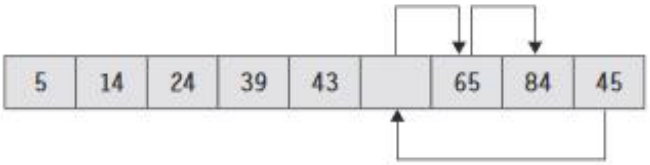
Cristian Pilaparita 1095211-2202112327

10.2.2. Ordenación por inserción

Este método consiste en insertar un elemento en el vector en una parte ya ordenada y comenzar de nuevo con los elementos restantes; debido a que generalmente lo utilizan por los jugadores de cartas, se le conoce también por el nombre de *método de la baraja*.

Así, por ejemplo, suponga que tiene la lista desordenada

Para insertar el elemento 45, habrá que insertarlo entre 43 y 65, lo que supone desplazar a la derecha todos aquellos números de valor superior a 45, es decir, saltar sobre 65 y 84.



El método se basa en comparaciones y desplazamientos sucesivos. El algoritmo de clasificación de un vector X para N elementos se realiza con un recorrido de todo el vector y la inserción del elemento correspondiente en el lugar adecuado. El recorrido se realiza desde el segundo elemento al *n-ésimo*.

```
desde i ← 2 hasta N hacer
   //insertar X[i] en el lugar
   //adecuado entre X[1]..X[i - 1])
fin_desde
```

Esta acción repetitiva —*insertar*— se realiza más fácilmente con la inclusión de un valor centinela o bandera (SW).

Pseudocódigo

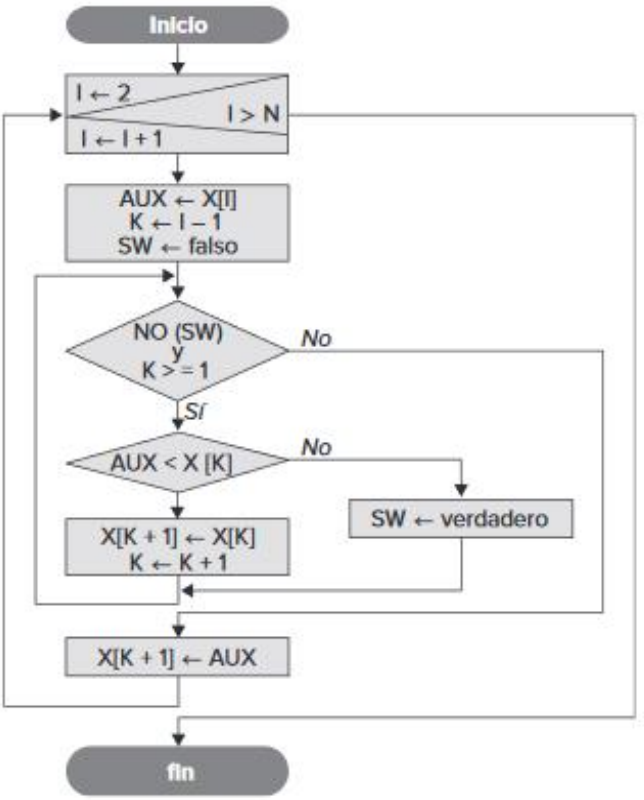
```
algoritmo clas_inserción1
//declaraciones
inicio
   .....
   //ordenación
   desde I ← 2 hasta N hacer
      AUXI ← X[I]
      K ← I - 1
      SW ← falso
      mientras no (SW) y (K >= 1) hacer
         si AUXI < X[K] entonces
            X[K+1] ← X[K]
            K ← K - 1
         si_no
            SW ← verdadero
         fin_si
      fin_mientras
      X[K+1] ← AUXI
   fin_desde
fin
```

Cristian Pilaquinta 1095211-2202112328

Algoritmo de inserción mejorado

El algoritmo de inserción directa se mejora fácilmente. Para ello se recurre a un método de búsqueda binaria —en lugar de una búsqueda secuencial— para encontrar más rápidamente el lugar de inserción. Este método se conoce como *inserción binaria*.

```
algoritmo clas_inserción_binaria
//declaraciones
inicio
   //...
   desde I ← 2 hasta N hacer
      AUX ← X[I]
      P ← 1    //primero
      U ← I - 1  //último
      mientras P <= U hacer
         C ← (P+U) div 2
         si AUX < X[C] entonces
            U ← C - 1
         si_no
            P ← C+1
         fin_si
      fin_mientras
      desde K ← I - 1 hasta P decremento 1 hacer
         X[K+1] ← X[K]
      fin_desde
         X[P] ← AUX
   fin_desde
fin
```



Cristian Pilaparita 1095211-2202112329

Número de comparaciones

El cálculo del número de comparaciones $\Gamma(n)$ que se realiza en el algoritmo de inserción se puede calcular fácilmente.

Consideraremos el elemento que ocupa la posición X en un vector de n elementos, en el que los $X-1$ elementos anteriores se encuentran ya ordenados ascendentemente por su clave; si la clave del elemento a insertar es mayor que las restantes, el algoritmo ejecuta sólo una comparación; si la clave es inferior a las restantes, el algoritmo ejecuta $X-1$ comparaciones. El número de comparaciones tiene por media $X/2$. Veamos los casos posibles.

Vector ordenado en origen

Comparaciones mínimas:

$$(n-1)$$

Vector inicialmente en orden inverso

Comparaciones máximas:

$$\frac{n(n-1)}{2}$$

ya que:

$$(n-1)+(n-2)+\ldots+3+2+1=\frac{(n-1)n}{2} \qquad \textit{es una progresión aritmética}$$

Comparaciones medias:

$$\frac{(n-1)+(n-1)n/2}{2}=\frac{n^2+n-2}{4}$$

otra forma de deducirlas sería:

$$Cmedias=\underbrace{\frac{n-1+1}{2}+\frac{n-2+1}{2}+\ldots+\frac{1+1}{2}}_{n-1 \text{ veces}}$$

y la suma de los términos de una progresión aritmética es:

$$Cmedias=(n-1)\frac{(n/2)+1}{2}=(n-1)\frac{n+2}{4}=\frac{n^2+2n-n-2}{4}=\frac{n^2+n-2}{4}$$

10.2.3. Ordenación por selección

Este método se basa en buscar el elemento menor del vector y colocarlo en primera posición. Luego se busca el segundo elemento más pequeño y se coloca en la segunda posición, y así sucesivamente. Los pasos sucesivos a dar son:

1. Seleccionar el elemento menor del vector de n elementos.
2. Intercambiar dicho elemento con el primero.
3. Repetir estas operaciones con los $n-1$ elementos restantes, seleccionando el segundo elemento; continuar con los $n-2$ elementos restantes hasta que sólo quede el mayor.

Un ejemplo aclarará el método.

Cristian Pilaparita 1095211-2202112329

EJEMPLO 10.2

Clasificar la siguiente lista de números en orden ascendente:

```
320   96   16   90   120   80   200   64
```

El método comienza buscando el número más pequeño, 16.

```
320   96   16   90   120   80   200   64
```

La lista nueva será:

```
16   96   320   90   120   80   200   64
```

A continuación, se busca el siguiente número más pequeño, 64, y se realizan las operaciones 1 y 2. La nueva lista sería:

```
16   64   320   90   120   80   200   96
```

Si se siguen realizando dos iteraciones se encontrará la siguiente línea:

```
16   64   80   90   120   320   200   96
```

No se realiza ahora ningún cambio, ya que el número más pequeño del vector `V[4], V[5], ..., V[8]` está ya en la posición más a la izquierda. Las sucesivas operaciones serán:

```
16   64   80  90   96    32   200   120
16   64   80  90   96   120   200   320
16   64   80  90   96   120   200   320
```

y se habrán terminado las comparaciones, ya que el último elemento debe ser el más grande y, por consiguiente, estará en la posición correcta.

Desarrollemos ahora el algoritmo para clasificar el vector `V` de `n` componentes `V[1], V[2], ..., V[n]` con este método. El algoritmo se presentará en etapas y lo desarrollaremos con un refinamiento por pasos sucesivos.

La tabla de variables que utilizaremos será:

`I, J`	*enteras* y se utilizan como índices del vector `V`
`X`	*vector* (array unidimensional)
`AUX`	variables auxiliar para intercambio
`N`	número de elementos del vector `V`

Nivel 1

```
inicio
   desde I ← 1 hasta N-1 hacer
      Buscar elemento menor de X[I], X[I+1], ..., X[N] e intercambiar con X[I]
   fin_desde
fin
```

Nivel 2

```
inicio
   I ← 1
   repetir
      Buscar elemento menor de X[I], X[I+1], ..., X[N] e intercambiar con X[I]
      I ← I+1
   hasta_que I = N
fin
```

La búsqueda e intercambio se realiza `N - 1` veces, ya que `I` se incrementa en 1 al final del bucle.

Cristian Pilapanta 1095211-2202112329

Nivel 3

Dividamos el bucle repetitivo en dos partes:

```
inicio
   I ←1
   repetir
      Buscar elemento más pequeño X[I], X[I+1], ..., X[N]
      //Supongamos que es X[K]
      Intercambiar X[K] y X[I]
   hasta_que I = N
fin
```

Nivel 4a

Las instrucciones "buscar" e "intercambiar" se refinan independientemente. El algoritmo con la estructura **repetir** es:

```
inicio
   //...
   I ← 1
   repetir
      AUXI ← X[I]   //AUXI representa el valor más pequeño
      K ← I         //K representa la posición
      J ← I
      repetir
         J ← J+1
         si X[J] < AUXI entonces
            AUXI ← X[J]    //actualizar AUXI
            K ← J          //K, posición
         fin_si
      hasta_que J = N   //AUXI = X[K] es ahora el más pequeño
      X[K] ← X[I]
      X[I] ← AUXI
      I ← I+1
   hasta_que I = N
fin
```

Nivel 4b

El algoritmo con la estructura **mientras**.

```
inicio
   //...
   I ← 1
   mientras I < N hacer
      AUXI ← X[I]
      K ← I
      J ← I
      mientras J < N hacer
         J ← J+1
         si X[J] < AUXI entonces
            AUXI ← X[J]
            K ← J
         fin_si
      fin_mientras
      X[K] ← X[I]
      X[I] ← AUXI
      I ← I + 1
   fin_mientras
fin
```

Cristian Pilaparita 1095211-2202112329

Nivel 4c

El algoritmo de ordenación con estructura **desde**.

```
inicio
   //...
   desde I ← 1 hasta N-1 hacer
      AUXI ← X[I]
      K ← I
      desde J ← I+1 hasta N hacer
         si X[J] < AUXI entonces
            AUXI ← X[J]
            K ← J
         fin_si
      fin_desde
      X[K] ← X[I]
      X[I] ← AUXI
   fin_desde
fin
```

10.2.4. Método de Shell

Es una mejora del método de inserción directa que se utiliza cuando el número de elementos a ordenar es grande. El método se denomina "Shell" —en honor de su inventor Donald Shell— y también método de *inserción* con incrementos decrecientes.

En el método de clasificación por inserción cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es más pequeño —por ejemplo—, es necesario ejecutar muchas comparaciones antes de colocarlo en su lugar definitivamente.

Shell modificó los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con eso se conseguía la clasificación más rápida. El método se basa en fijar el tamaño de los saltos constantes, pero de más de una posición.

Supongamos un vector de elementos:

```
4    12    16    24    36    3
```

en el método de inserción directa, los saltos se hacen de una posición en una posición y se necesitarán cinco comparaciones. En el método de Shell, si los saltos son de dos posiciones, se realizan tres comparaciones.

```
4    12    16    24    36    3
```

El método se basa en tomar como salto `N/2` (siendo `N` el número de elementos) y luego se va reduciendo a la mitad en cada repetición hasta que el salto o distancia vale 1.

Considerando la variable *salto*, se tendría para el caso de un determinado vector `X` los siguientes recorridos:

```
Vector X    [X[1], X[2], X[3], ..., X[N]]
Vector X1   [X[1], X[1+salto], X[2+salto], ...]
Vector XN   [salto1, salto2, salto3, ...]
```

EJEMPLO 10.3

Deducir las secuencias parciales de clasificación por el método de Shell para ordenar en ascendente la lista o vector:

```
6,    1,    5,    2,    3,    4,    0
```

Cristian Pitaparta 1095211-2202112330

Solución

Recorrido	*Salto*	*Lista reordenada*	*Intercambio*
1	3	2,1,4,0,3,5,6	(6,2), (5,4), (6,0)
2	3	0,1,4,2,3,5,6	(2,0)
3	3	0,1,4,2,3,5,6	Ninguno
4	1	0,1,2,3,4,5,6	(4,2), (4,3)
5	1	0,1,2,3,4,5,6	Ninguno

Sea un vector X:

```
X[1], X[2], X[3], ..., X[N]
```

y consideremos el primer salto a dar que tendrá un valor de:

$$\frac{N}{2}$$

por lo que, para redondear, se tomará la parte entera:

```
N DIV 2
```

y se iguala a salto:

```
salto = N div 2
```

El algoritmo resultante será:

```
algoritmo shell
const
   n = 50
tipo
   array[1..n] de entero:lista
var
   lista : L
   entero : k, i, j, salto
inicio
   llamar_a llenar(L)              // llenado de la lista
   salto ← N DIV 2
   mientras salto > 0 hacer
      desde i ← (salto + 1) hasta n hacer
         j ← i - salto
         mientras j > 0 hacer
            k ← j + salto
            si L[j] <= L[k] entonces
               j ← 0
            si_no
               llamar_a intercambio L[j], L[k]
            fin_si
            j ← j - salto
         fin_mientras
      fin_desde
      salto ← ent ((1 + salto)/2)
   fin_mientras
fin
```

Cristian Pilaparta 1095211-2202112330

10.2.5. Método de ordenación rápida (*quicksort*)

El método de *ordenación rápida* (*quicksort*) para ordenar o clasificar un vector o lista de elementos (array) se basa en el hecho de que es más rápido y fácil de ordenar dos listas pequeñas que una lista grande. Se denomina método de ordenación rápida porque, en general, puede ordenar una lista de datos mucho más rápidamente que cualquiera de los métodos de ordenación ya estudiados.

Este método se debe a Hoare y se basa en la estrategia típica de *divide y vencerás* (*divide and conquer*); la lista a clasificar almacenada en un vector o array se divide (*parte*) en dos sublistas: una con todos los valores menores o iguales a un cierto valor específico y otra con todos los valores mayores que ese valor. El valor elegido puede ser cualquier valor arbitrario del vector, en ordenación rápida se llama a este valor *pivote*.

El primer paso es dividir la lista original en dos sublistas o subvectores y un valor de separación; así, el vector `V` se divide en tres partes:

- Subvector `VI`, que contiene los valores inferiores o iguales.
- El elemento de separación.
- Subvector `VD`, que contiene los valores superiores o iguales.

Los subvectores `VI` y `VD` no están ordenados, excepto en el caso de reducirse a un elemento. Consideremos la lista de valores.

```
18   11   27   13   9   4   16
```

Se elige un pivote, 13; se recorre la lista desde el extremo izquierdo y se busca un elemento mayor que 13 (se encuentra el 18). A continuación, se busca desde el extremo derecho un valor menor que 13 (se encuentra el 4).

```
18   11   27   13   9   4   16
```

Se intercambian estos dos valores y se produce la lista:

```
4   11   27   13   9   18   16
```

Se sigue recorriendo el vector por la izquierda y se localiza el 27, y a continuación otro valor bajo se encuentra a la derecha (el 9). Intercambiar estos dos valores y se obtiene:

```
4   11   9   13   27   18   16
```

Al intentar este proceso una vez más, se encuentra que las exploraciones de los dos extremos vienen juntas, sin encontrar ningún futuro valor que esté "fuera de lugar". En este punto se conoce que todos los valores a la derecha son mayores que todos los valores a la izquierda del pivote. Se ha realizado una partición en la lista original, que se ha quedado dividida en dos listas más pequeñas:

```
4   11   9   [13]   2   18   16
```

Ninguna de las listas está ordenada; sin embargo, basados en los resultados de esta primera partición, se pueden ordenar ahora las dos particiones independientemente. Esto es, si ordenamos la lista:

```
4   11   9
```

en su posición, y la lista:

```
27   18   16
```

de igual forma, completa estará ordenada:

```
4   9   11   13  16   18   27
```

El procedimiento de ordenación supone, en primer lugar, una partición de la lista.

Cristian Pilapaña 1095211-2202112330

EJEMPLO 10.4

Utilizando el procedimiento de ordenación rápida, dividir la lista de enteros en dos sublistas para, posteriormente, poder clasificarlas.

50 30 20 80 90 70 95 85 10 15 75 25

Se elige como pivote el número 50.

Los valores 30, 20, 10, 15 y 25 son más pequeños que 50 y constituirán la primera lista, y 80, 90, 70, 95, 85 y 75 se sitúan en la segunda lista. Se recorre la lista desde la izquierda para encontrar el primer número mayor que 50 y desde la derecha el primero menor que 50.

50 30 20 80 90 70 95 85 10 15 75 25

se localizan los dos números 80 y 25 y se intercambian

50 30 20 25 90 70 95 85 10 15 75 80

A continuación, se reanuda la búsqueda desde la derecha para un número menor que 50, y desde la izquierda para un número mayor que 50.

50 30 20 25 90 70 95 85 10 15 75 80

Estos recorridos localizan los números 15 y 90, que se intercambian

50 30 20 25 15 70 95 85 10 90 75 80

Las búsquedas siguientes localizan 10 y 70.

50 30 20 25 15 70 95 85 10 90 75 80

El intercambio proporciona

50 30 20 25 15 10 95 85 70 90 75 80

Cuando se reanuda la búsqueda desde la derecha para un número menor que 50, localizamos el valor 10 que se encontró en la búsqueda de izquierda a derecha. Se señala el final de las dos búsquedas y se intercambian 50 y 10.

10 30 20 25 15 [50] 95 85 70 75 80

Lista de números < 50 *Lista de números > 50*

Cristian Pilaparra 1095211-2202112330

Algoritmos

El algoritmo de ordenación rápida se basa esencialmente en un algoritmo de división o partición de una lista. El método consiste en explorar desde cada extremo e intercambiar los valores encontrados; un primer intento de algoritmo de partición es:

```
algoritmo partición
inicio
   establecer x al valor de un elemento arbitrario de la lista
   mientras división no esté terminada hacer
      recorrer de izquierda a derecha para un valor >= x
      recorrer de derecha a izquierda para un valor =< x
      si los valores localizados no están ordenados entonces
         intercambiar los valores
      fin_si
   fin_mientras
fin
```

La lista que se desea partir es `A[1], A[2], ..., A[n]`; los índices que representan los extremos izquierdo y derecho de la lista son `L` y `R`. En el refinamiento del algoritmo se elige un valor arbitrario `X`, suponiendo que el valor central de la lista es tan bueno como cualquier elemento arbitrario. Los índices `i, j` exploran desde los extremos. Un refinamiento del algoritmo anterior, que incluye mayor número de detalles es el siguiente:

```
algoritmo partición
   llenar (A)
   i ← L
   j ← R
   x ← A ((L+R) div 2)
   mientras i =< j hacer
      mientras A[i] < x hacer
         i ← i+1
      fin_mientras
      mientras A[j] > x hacer
         j ← j-1
      fin_mientras
      si i =< j entonces
         llamar_a intercambiar (A[i], a[j])
         i ← i+1
         j ← j-1
      fin_si
   fin_mientras
fin
```

En los bucles externos y la sentencia **si** la condición utilizada es `i =< j`. Puede parecer que `i < j` funcionan de igual modo en ambos lugares; de hecho, se puede realizar la partición con cualquiera de las condiciones. Sin embargo, si se utiliza la condición `i < j`, podemos terminar la partición con dos casos distintos, los cuales pueden diferenciarse antes de que podamos realizar divisiones futuras; por ejemplo, la lista:

```
1   7   7   9   9
```

y la condición `i < j` terminará con `i = 3, j = 2` y las dos particiones son `A[L]...A[j]` y `A[i]...A[R]`. Sin embargo, para la lista:

```
1   1   7   9   9
```

y la condición `i < j` terminaremos con `i = 3, j = 3` y las dos particiones se solapan.

El uso de la condición `i <= j` produce también resultados distintos para estos ejemplos. La lista:

```
1   7   7   9   9
```

Cristian Pilaparita 1095211-2202112330

y la condición i <= j se termina con i = 3, j = 2 como antes. Para la lista 1, 1, 7, 9, 9 y la condición i =< j se termina con i = 4, j = 2. En ambos casos las particiones que requieren ordenación posterior son A[L]...A[j] y A[i]...A[R].

En los bucles **mientras** internos, la igualdad se omite de las condiciones; la razón es que el valor de partición actúe como *centinela* para detectar las exploraciones.

En nuestro ejemplo se ha tomado como valor de partición o pivote el elemento cuya posición inicial es el elemento central; éste no es generalmente el caso. El ejemplo de la clasificación de la lista ya citado:

```
50   30   20   80   90   70   95   85   10   15   75   25
```

utilizaba como pivote el primer elemento.

El algoritmo de ordenación rápida en el caso de que el elemento pivote sea el primer elemento se muestra a continuación:

```
algoritmo partición2
//lista a evaluar de 10 elementos
//IZQUIERDO, índice de búsqueda (recorrido) desde la izquierda
//DERECHO, índice de búsqueda desde la derecha

inicio
   llenar (X)
   //inicializar índice para recorridos desde la izquierda y derecha
   IZQUIERDO ← ALTO  //ALTO parámetro que indica principio de la sublista
   DERECHO ← BAJO    //BAJO parámetro que indica final de la sublista
   A <-  X[1]
   //realizar los recorridos
   mientras IZQUIERDO <= DERECHO hacer
   //búsqueda o recorrido desde la izquierda
      mientras (X[IZQUIERDO] < A) Y (IZQUIERDO < BAJO)
         IZQUIERDO ← IZQUIERDO + 1
      fin_mientras
      mientras X[DERECHO] > A y (DERECHO > ALTO)
         DERECHO ← DERECHO - 1
      fin_mientras
      //intercambiar elemento
      si IZQUIERDO <= DERECHO entonces
         AUXI <-X[IZQUIERDO]
         X[IZQUIERDO] ← X[DERECHO]
         X[DERECHO] ← AUXI
         IZQUIERDO ← IZQUIERDO + 1
         DERECHO ← DERECHO - 1
      fin_si
   fin_mientras
   //fin búsqueda: situar elemento seleccionado en su posición
   si IZQUIERDO < BAJO+1 entonces
      AUXI← X [DERECHO]
      X [DERECHO] ← X [1]
      X [1] ← AUXI
   si_no
      AUXI ← X [BAJO]
      X [BAJO] ← X[1]
      X[1] ← AUXI
   fin
fin
```

Cristian P. Laparte 1095211-2202112330

10.3. Búsqueda

La recuperación de información, como ya se ha comentado, es una de las aplicaciones más importantes de las computadoras.

La *búsqueda* (*searching*) de información está relacionada con las tablas para consultas (*lookup*). Estas tablas contienen una cantidad de información que se almacena en forma de listas de parejas de datos. Por ejemplo, un diccionario con una lista de palabras y definiciones, un catálogo con una lista de libros de informática, una lista de estudiantes y sus notas; un índice con títulos y contenido de los artículos publicados en una determinada revista, etcétera. En todos estos casos con frecuencia es necesario buscar un elemento en una lista.

Una vez que se encuentra el elemento, la identificación de su información correspondiente es un problema menor. Por consiguiente, nos centraremos en el proceso de búsqueda. Supongamos que se desea buscar en el vector `X[1]... X[n]`, que tiene componentes numéricos, para ver si contiene o no un número dado `T`.

Si en vez de tratar sobre vectores se desea buscar información en un archivo, debe realizarse la búsqueda a partir de un determinado campo de información denominado *campo clave*. Así, en el caso de los archivos de empleados de una empresa, el campo clave puede ser el número de DNI o los apellidos.

La búsqueda por claves para localizar registros es, con frecuencia, una de las acciones que mayor consumo de tiempo conlleva y, por consiguiente, el modo en que los registros están dispuestos y la elección del modo utilizado para la búsqueda pueden redundar en una diferencia sustancial en el rendimiento del programa.

El problema de búsqueda cae naturalmente dentro de los dos casos típicos ya tratados. Si existen muchos registros, puede ser necesario almacenarlos en archivos de disco o cinta, externo a la memoria de la computadora. En este caso se llama *búsqueda externa*; en el otro caso, los registros que se buscan se almacenan por completo dentro de la memoria de la computadora y se denomina *búsqueda interna*.

En la práctica, la búsqueda se refiere a la operación de encontrar la posición de un elemento entre un conjunto de elementos dados: lista, tabla o fichero, ejemplos típicos de búsqueda son localizar nombre y apellidos de un alumno, localizar números de teléfono de una agenda, etcétera. Existen diferentes algoritmos de búsqueda, el elegido depende de la forma en que se encuentren organizados los datos.

La operación de búsqueda de un elemento *N* en un conjunto de elementos consiste en:

- Determinar si *N* pertenece al conjunto y, en ese caso, indicar su posición en él.
- Determinar si *N* no pertenece al conjunto.

Los métodos más usuales de búsqueda son:

- *Búsqueda secuencial o lineal.*
- *Búsqueda binaria.*
- *Búsqueda por transformación de claves (hash).*

10.3.1. Búsqueda secuencial

Supongamos una lista de elementos almacenados en un vector (array unidimensional). El método más sencillo de buscar un elemento en un vector es explorar secuencialmente el vector o, dicho en otras palabras, *recorrer el vector* desde el primer elemento al último. Si se encuentra el elemento buscado, visualizar un mensaje similar a `'Fin de búsqueda'`; en caso contrario, visualizar un mensaje similar a `'Elemento no existe en la lista'`.

En otras palabras, la búsqueda secuencial compara cada elemento del vector con el valor deseado, hasta que éste encuentra o termina de leer el vector completo.

La búsqueda secuencial no requiere ningún registro por parte del vector y, por consiguiente, no necesita estar ordenado. El recorrido del vector se realizará normalmente con estructuras repetitivas.

EJEMPLO 10.5

Se tiene un vector `A` *que contiene* `n` *elementos numéricos* `(n) >= 1(A[1], A[2], A[3], ..., A[n])` *y se desea buscar un elemento dado* `t`. *Si el elemento* `t` *se encuentra, visualizar un mensaje* `'Elemento encontrado'` *y otro que diga* `'posicion = '`.

Cristian Pilponi 1/8/21-20211230

Si existen n elementos, se requerirán como media n/2 comparaciones para encontrar un determinado elemento. En el caso más desfavorable se necesitarán n comparaciones.

Método 1

```
algoritmo búsqueda_secuencial_1
   //declaraciones
inicio
   llenar (A,n)
   leer(t)
   //recorrido del vector
   desde i ← 1 hasta n hacer
      si A[i] = t entonces
         escribir('Elemento encontrado')
         escribir('en posición', i)
      fin_si
   fin_desde
fin
```

Método 2

```
algoritmo búsqueda_secuencial_2
   //...
inicio
   llenar (A,n)
   leer(t)
   i ← 1
   mientras (A[i] <> t) y (i =< n) hacer
      i ← i + 1
      //este bucle se detiene bien con A[i] = t o bien con i > n
   fin_mientras
   si A[i] = t entonces      //condición de parada
      escribir('El elemento se ha encontrado en la posición', i)
   si_no                     //recorrido del vector terminado
      escribir('El número no se encuentra en el vector')
   fin_si
fin
```

Este método no es completamente satisfactorio, ya que si t no está en el vector A, i toma el valor n + 1 y la comparación:

```
A[i] <> t
```

producirá una referencia al elemento A[n + 1], que presumiblemente no existe. Este problema se resuelve sustituyendo i =< n por i < n en la instrucción **mientras**, es decir, modificando la instrucción anterior **mientras** por:

```
mientras (A[i] <> t) y (i < n) hacer
```

Método 3

```
algoritmo búsqueda_secuencial_3
   //...
inicio
   llenar (A,n)
   leer(t)
   i ← 1
```

Cristian Pilipanta 1095211-2202112330

```
    mientras (A[i] <> t) y (i < n) hacer
        i ← i+1
        //este bucle se detiene cuando A[i] = t o i >= n
    fin_mientras
    si A[i] = t entonces
        escribir('El número deseado está presente y ocupa el lugar',i)
    si_no
        escribir(t, 'no existe en el vector')
    fin_si
fin
```

Método 4

```
algoritmo búsqueda_secuencial_4
    //...
inicio
    llamar_a llenar(A,n)
    leer(t)
    i ← 1
    mientras i <= n hacer
        si t = A[i] entonces
            escribir('Se encontró el elemento buscado en la posición',i)
            i ← n + 1
        si_no
            i ← i+1
        fin_si
    fin_mientras
fin
```

Cristian Pilaparta 1095211-2202112330

Búsqueda secuencial con centinela

Una manera muy eficaz de realizar una búsqueda secuencial consiste en modificar los algoritmos anteriores utilizando un elemento centinela, el cual se agrega al final del vector. El valor del elemento centinela, `A[n + 1]`, es el del argumento. Su propósito es significar que la búsqueda siempre tendrá éxito. El elemento `A[n + 1]` sirve como centinela y se le asigna el valor de `t` antes de iniciar la búsqueda. En cada paso se evita la comparación de `i` con `N` y, por consiguiente, este algoritmo será preferible a los métodos anteriores, concretamente el método 4. Si el índice alcanzase el valor `n + 1`, supondría que el argumento no pertenece al vector original y en consecuencia la búsqueda no tiene éxito.

Método 5

```
algoritmo búsqueda_secuencial_5
    //declaraciones
inicio
    llenar(A,n)
    leer(t)
    i ← 1
    A[n + 1] ← t
    mientras A[i] <> t hacer
        i ← i + 1
    fin_mientras
    si i = n + 1 entonces
        escribir('No se ha encontrado elemento')
```

```
    si_no
       escribir('Se ha encontrado el elemento')
    fin_si
fin
```

Una variante del método 5 es utilizar una variable lógica (interruptor o *switch*), que represente la existencia o no del elemento buscado.

Localizar si el elemento `t` existe en una lista `A[i]`, donde `i` varía desde 1 a `n`.

En este ejemplo se trata de utilizar una variable lógica `ENCONTRADO` para indicar si existe o no el elemento de la lista.

Método 6

```
algoritmo búsqueda_secuencia_6
   //declaraciones
inicio
   llenar (A,n)
   leer(t)
   i ← 1
   ENCONTRADO ← falso
   mientras (no ENCONTRADO) y (i =< n) hacer
      si A[i] = t entonces
         ENCONTRADO ← verdadero
      fin_si
      i ← i + 1
   fin_mientras
   si ENCONTRADO entonces
      escribir('El número ocupa el lugar', i - 1)
   si_no
      escribir('El número no está en el vector')
   fin_si
fin
```

Nota

De todas las versiones anteriores, tal vez la más adecuada sea la incluida en el método 6. Entre otras razones, debido a que el bucle **mientras** engloba las acciones que permiten explorar el vector, bien hasta que *t* se encuentre o bien cuando se alcance el final del vector.

Método 7

```
algoritmo búsqueda_secuencia_7
   //declaraciones
inicio
   llenar (A,n)
   leer(t)
   i ← 1
   ENCONTRADO ← falso
   mientras i =< n hacer
      si A[i] = t entonces
         ENCONTRADO ← verdad
         escribir ('El número ocupa el lugar'; i)
      fin_si
```

Cristian Pilaparta 1095211-2202112330

```
      i ← i + 1
   fin_mientras
   si_no (ENCONTRADO) entonces
      escribir('El número no está en el vector')
   fin_si
fin
```

Método 8

```
algoritmo búsqueda_secuencial_8
   //declaraciones
inicio
   llenar (A,n)
   ENCONTRADO ← falso
   i ← 0
   leer(t)
   repetir
      i ← i+1
      si A[i] = t entonces
         ENCONTRADO ← verdad
      fin_si
   hasta_que ENCONTRADO o (i = n)
fin
```

Método 9

```
algoritmo búsqueda_secuencial_9
   //declaraciones
inicio
   llenar (A,n)
   ENCONTRADO ← falso
   leer(t)
   desde i ← 1 hasta i ← n hacer
      si A[i] = t entonces
         ENCONTRADO ← verdad
      fin_si
   fin_desde
   si ENCONTRADO entonces
      escribir('Elemento encontrado')
   si_no
      escribir('Elemento no encontrado')
   fin_si
fin
```

Consideraciones sobre la búsqueda lineal

El método de búsqueda lineal tiene el inconveniente del consumo excesivo de tiempo en la localización del elemento buscado. Cuando éste no se encuentra en el vector, se verifican o comprueban sus *n* elementos.

En los casos en que el elemento se encuentra en la lista, el número podrá ser el primero, el último o alguno comprendido entre ambos; se puede suponer que el número medio de comprobaciones o comparaciones a realizar es de $(n+1)/2$ (aproximadamente igual a la mitad de los elementos del vector).

La búsqueda secuencial o lineal no es el método más eficiente para vectores con un gran número de elementos. En estos casos, el método más idóneo es el de *búsqueda binaria*, que presupone una ordenación previa en los ele-

Cristian Pizarro 1095211-2003112300

mentos del vector. Este caso suele ser muy utilizado en numerosas facetas de la vida diaria. Un ejemplo de ello es la búsqueda del número de un abonado en una guía telefónica; normalmente no se busca el nombre en orden secuencial, sino que se busca en la primera o segunda mitad de la guía; una vez en esa mitad, se vuelve a tantear a una de sus dos submitades, y así sucesivamente se repite el proceso hasta que se localiza la página correcta.

10.3.2. Búsqueda binaria

En una búsqueda secuencial se comienza con el primer elemento del vector y se busca en él hasta que se encuentra el elemento deseado o se alcanza su final; aunque éste puede ser un método adecuado para pocos datos, se necesita una técnica más eficaz para conjuntos grandes de datos.

Si el número de elementos del vector es grande, el algoritmo de búsqueda lineal se ralentizaría en tiempo de un modo considerable. Por ejemplo, si tuviéramos que consultar un nombre en la guía telefónica de una gran ciudad como Madrid, con una cifra aproximada de tres millones de abonados, el tiempo de búsqueda —según el nombre— se podría eternizar. Naturalmente, las personas que viven en esa gran ciudad nunca utilizarán un método de búsqueda secuencial, sino un método que se basa en la división sucesiva del espacio ocupado por el vector en sucesivas mitades, hasta encontrar el elemento buscado.

Si los datos que se buscan están clasificados en un determinado orden, el método citado anteriormente se denomina *búsqueda binaria*; para localizar el valor deseado esta búsqueda utiliza el método denominado *divide y vencerás*. Con este método se examina primero el elemento central de la lista, si éste es el elemento buscado, entonces la búsqueda ha terminado; en caso contrario, se determina si está en la primera o la segunda mitad de la lista y, a continuación, se repite este proceso utilizando el elemento central de esa sublista. Supongamos la lista:

```
1231
1473
1545
1834
1892
1898        elemento central
1983
2005
2446
2685
3200
```

Si está buscando el elemento 1983, se examina el número central, 1898, en la sexta posición. Ya que 1983 es mayor que 1898, se desprecia la primera sublista y nos centramos en la segunda:

```
1983
2005
2446        elemento central
2685
3200
```

El número central de esta sublista es 2446 y el elemento buscado es 1983, menor que 2446; eliminamos la segunda sublista y nos queda:

```
1983
2005
```

Como no hay término central, elegimos el término inmediatamente anterior, 1983, que es el buscado. Se han necesitado tres comparaciones, mientras que la búsqueda secuencial hubiese necesitado siete.

La búsqueda binaria se utiliza en vectores ordenados y se basa en la constante división del espacio de búsqueda (recorrido del vector). Como se ha comentado, se comienza comparando el elemento que se busca, no con el

Cristian P[illegible] [illegible]

primer elemento, sino con el elemento central. Si el elemento buscado —*t*— es menor que el elemento central, entonces *t* deberá estar en la mitad izquierda o inferior del vector; si es mayor que el valor central, deberá estar en la mitad derecha o superior, y si es igual al valor central, se habrá encontrado el elemento buscado.

El funcionamiento de la búsqueda binaria en un vector de enteros se ilustra en la figura 10.3 para dos búsquedas: *con éxito* (localizado el elemento) y *sin éxito* (no encontrado el elemento).

El proceso de búsqueda debe terminar normalmente conociendo si la búsqueda *ha tenido éxito* (se ha encontrado el elemento) o bien *no ha tenido éxito* (no se ha encontrado el elemento) y normalmente se deberá devolver la posición del elemento buscado dentro del vector.

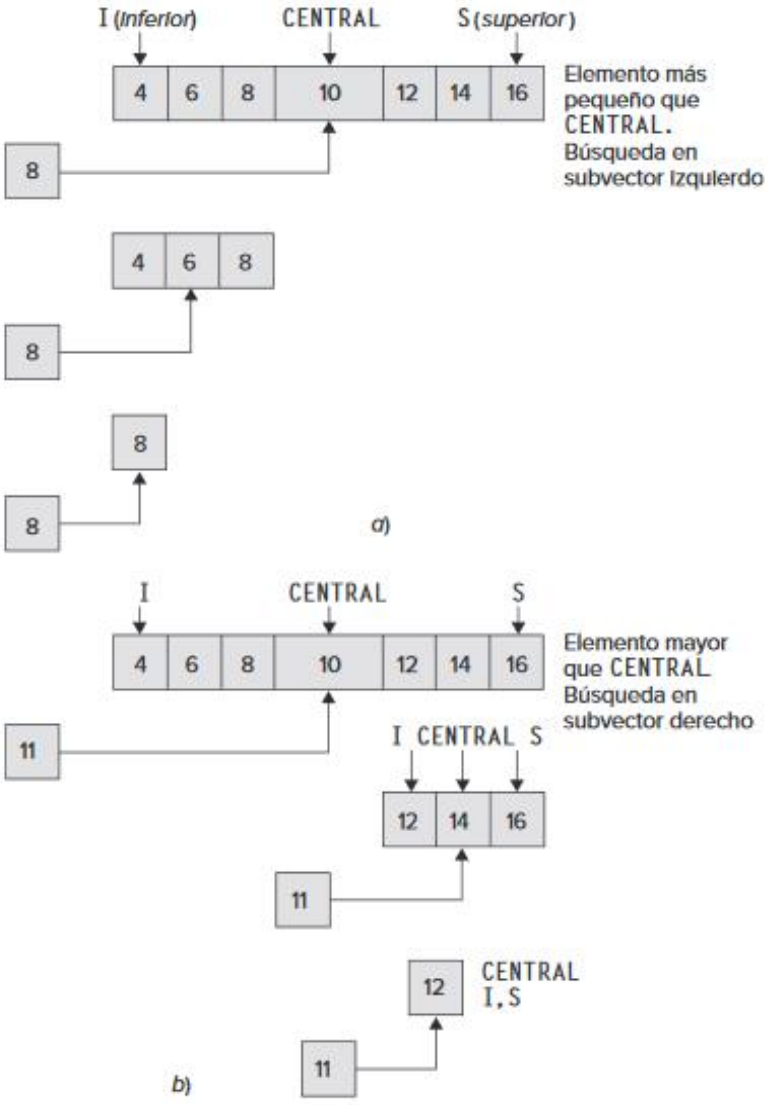


Figura 10.3. Ejemplo de búsqueda binaria: *a*) con éxito, *b*) sin éxito.

EJEMPLO 10.6

Realizar el algoritmo de búsqueda binaria para encontrar un elemento K *en una lista de elementos* X_1, X_2, . . . , X_n *previamente clasificados en orden ascendente.*

El array o vector *X* se supone ordenado en orden creciente si los datos son numéricos, o alfabéticamente si son caracteres. Las variables `BAJO`, `CENTRAL`, `ALTO` indican los límites inferior, central y superior del intervalo de búsqueda.

Cristian Pilaparita 1095211-2202112330

```
algoritmo búsqueda_binaria
   //declaraciones
inicio
   //llenar (X,N)
   //ordenar (X,N)
   leer(K)
   //inicializar variables
   BAJO ← 1
   ALTO ← N
   CENTRAL ← ent ((BAJO + ALTO) / 2)
   mientras (BAJO =< ALTO) y (X[CENTRAL] <> K) hacer
      si K < X[CENTRAL] entonces
         ALTO ← CENTRAL - 1
      si_no
         BAJO ← CENTRAL + 1
      fin_si
      CENTRAL ← ent ((BAJO + ALTO) / 2)
   fin_mientras
   si K = X[CENTRAL] entonces
      escribir('Valor encontrado en', CENTRAL)
   si_no
      escribir('Valor no encontrado')
   fin_si
fin
```

EJEMPLO 10.7

Se dispone de un vector tipo carácter NOMBRE *clasificado en orden ascendente y de* N *elementos. Realizar el algoritmo que efectúe la búsqueda de un nombre introducido por el usuario.*

La variable N indica cuántos elementos existen en el array.
ENCONTRADO es una variable lógica que detecta si se ha localizado el nombre buscado.

```
algoritmo búsqueda_nombre
{inicializar todas las variables necesarias}
{NOMBRE       array de caracteres
N             número de nombres del array NOMBRE
ALTO          puntero al extremo superior del intervalo
BAJO          puntero al extremo inferior del intervalo
CENTRAL       puntero al punto central del intervalo
X             nombre introducido por el usuario
ENCONTRADO    bandera o centinela}
inicio
   llenar (NOMBRE, N)
   leer (X)
   BAJO ← 1
   ALTO ← N
   ENCONTRADO ← falso
   mientras (no ENCONTRADO) y (BAJO =< ALTO) hacer
      CENTRAL ← ent (BAJO+ALTO) / 2
      //verificar nombre central en este intervalo
      si NOMBRE[CENTRAL] = X entonces
         ENCONTRADO ← verdad
      si_no
```

Cristian Pilaparita 1095211-2202112330

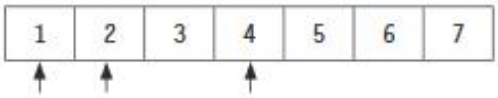
```
            si NOMBRE[CENTRAL] > X entonces
               ALTO ← CENTRAL - 1
            si_no
               BAJO ← CENTRAL + 1
            fin_si
         fin_si
      fin_mientras
      si ENCONTRADO entonces
         escribir('Nombre encontrado')
      si_no
         escribir('Nombre no encontrado')
      fin_si
   fin
```

Análisis de la búsqueda binaria

La búsqueda binaria es un método eficiente siempre que el vector esté ordenado. En la práctica esto suele suceder, pero no siempre. Por esta razón, la búsqueda binaria exige una ordenación previa del vector.

Para poder medir la velocidad de cálculo del algoritmo de búsqueda binaria se deberán obtener el número de comparaciones que realiza el algoritmo.

Consideremos un vector de siete elementos ($n = 7$). El número 8 ($N + 1 = 8$) se debe dividir en tres mitades antes de que se alcance 1; es decir, se necesitan tres comparaciones.



El medio matemático de expresar estos números es:

$$3 = log_2 (8)$$

en general, para n elementos:

$$K = log_2 (n + 1)$$

Recuerde que $log_2 (8)$ es el exponente al que debe elevarse 2 para obtener 8. Es decir, 3, ya que $2^3 = 8$.

Si $n + 1$ es una potencia de 2, entonces $log_2 (n + 1)$ será un entero. Si $n + 1$ no es una potencia de 2, el valor del logaritmo se redondea hasta el siguiente entero. Por ejemplo, si n es 12, entonces K será 4, ya que $log_2 (13)$ (que está entre 3 y 4) se redondeará hasta 4 (2^4 es 16).

En general, en el mejor de los casos se realizará una comparación y, en el peor de los casos, se realizarán $log_2 (n + 1)$ comparaciones.

Como término medio, el número de comparaciones es:

$$\frac{1 + \log_2(n + 1)}{2}$$

Esta fórmula se puede reducir para el caso de que n sea grande a:

$$\frac{\log_2(n + 1)}{2}$$

Para poder efectuar una comparación entre los métodos de búsqueda lineal y búsqueda binaria, realicemos los cálculos correspondientes para diferentes valores de n.

`n = 100` En la *búsqueda secuencial* se necesitarán

$$\frac{100 + 1}{2}$$ **50 comparaciones**

Cristian Pilaparita 10952111-2202112330

En la *búsqueda binaria* $\log_2 (100) = 6...$

$\log_2 (100) = x$ donde $2^x = 100$ y $x = 6...$:

$27 = 128 > 100$ **7 comparaciones**

$n = 1000000$ En la *búsqueda secuencial:*

$\frac{1000000 + 1}{2}$ **500000 comparaciones**

En la *búsqueda binaria* $\log_2 (1000000) = x$

$2^x = 1000000$ donde $x = 20$ y $2^{20} > 1000000$

20 comparaciones

Como se observa en los ejemplos anteriores, el tiempo de búsqueda es muy pequeño, aproximadamente siete comparaciones para 100 elementos y veinte para 1000000. (*Compruebe que para 1000 elementos se requiere un máximo de diez comparaciones.*)

La búsqueda binaria tiene, sin embargo, inconvenientes a resaltar:

El *vector debe estar ordenado* y *el almacenamiento de un vector ordenado* suele plantear problemas en las inserciones y eliminaciones de elementos. (En estos casos será necesario utilizar listas enlazadas o árboles binarios, capítulos 12 y 13.)

La tabla 10.2 compara la eficiencia de la búsqueda lineal y binaria para diferentes valores de *n*. Como se observa en dicha tabla, la ventaja del método de búsqueda binaria aumenta a medida que *n* aumenta.

Cristian Pilapanta 1095211-2202112330

Tabla 10.2. Eficiencia de las búsquedas lineal y binaria.

Búsqueda secuencial		Búsqueda binaria
Número de comparaciones		Número máximo de comparaciones
n	Elemento no localizado	Elemento no localizado
7	7	3
100	100	7
1.000	1.000	10
1.000.000	100.000	20

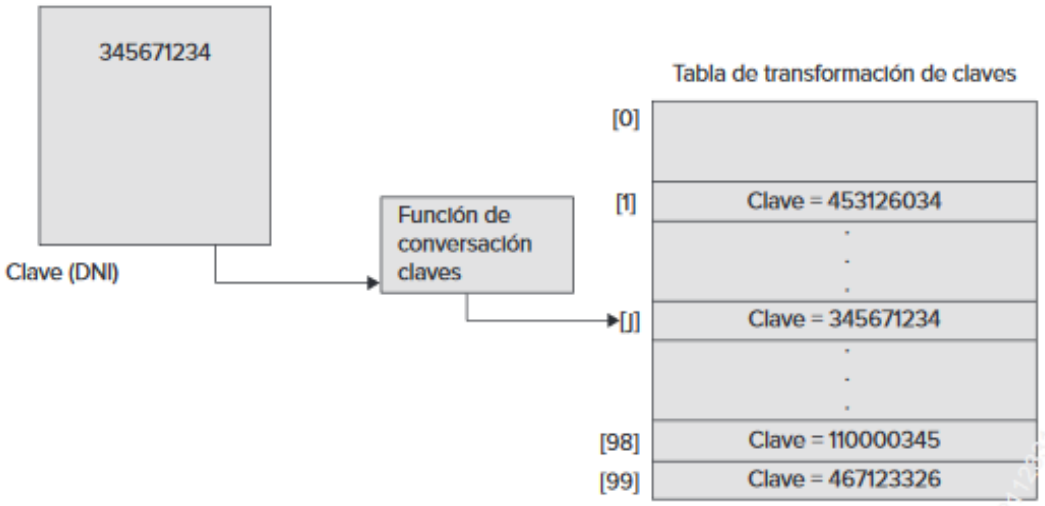
10.3.3. Búsqueda mediante transformación de claves (*hashing*)

La búsqueda binaria proporciona un medio para reducir el tiempo requerido para buscar en una lista; sin embargo, este método, exige que los datos estén ordenados. Existe otro método que pueden aumentar la velocidad de búsqueda y los datos no necesitan estar ordenados, este método se conoce como transformación de claves (clave-dirección) o *hashing*.

El método de transformación de claves consiste en convertir la clave dada (numérica o alfanumérica) en una dirección (índice) dentro del array. La correspondencia entre las claves y la dirección en el medio de almacenamiento o en el array se establece por una función de conversión (función o *hash*).

Así, por ejemplo, en el caso de una lista de empleados (100) de una pequeña empresa, si cada uno tiene un número de identificación (clave) del 1 al 100, evidentemente puede existir una correspondencia directa entre la clave y la dirección definida en un vector o array de 100 elementos.

Supongamos ahora que el campo clave de estos registros o elementos es el número del DNI o de seguridad social, que contenga nueve dígitos. Si se desea mantener en un array todo el rango posible de valores, se necesitarán 10^10 elementos en la tabla de almacenamiento, cantidad difícil de tener disponibles en memoria central, aproximadamente 1000000000 de registros o elementos. Si el vector o archivo sólo tiene 100, 200 o 1000 empleados, ¿cómo hacer para introducirlos en memoria por el campo clave DNI? Para hacer uso de la clave DNI como un índice en la tabla de búsqueda, se necesita un medio para convertir el campo clave en una dirección o índice más pequeño. En la figura se presenta un diagrama de cómo realizar la operación de conversión de una clave grande en una tabla pequeña.



Los registros o elementos del campo clave no tienen por qué estar ordenados de acuerdo con los valores del campo clave, como estaban en la búsqueda binaria; por ejemplo, el registro del campo clave 345671234 estará almacenado en la tabla de transformación de claves (array) en una posición determinada, es decir, 75.

La función de transformación de clave, $H(k)$ convierte la clave (k) en una dirección (d).

Imaginemos que las claves fueran nombres o frases de hasta 16 letras, que identifican a un conjunto de un millar de personas. Existirán 26^16 combinaciones posibles de claves que se deben transformar en 103 direcciones o índices posibles. La función H es, por consiguiente, una función de paso o conversión de múltiples claves a direcciones. Dada una clave k, el primer paso en la operación de búsqueda es calcular su índice asociado $d \leftarrow H(k)$ y el segundo paso —evidentemente necesario— es verificar *sí* o *no* el elemento con la clave k es identificado verdaderamente por d en el array T; es decir, para verificar si la clave `T[H(K)] = K` se deben considerar dos preguntas:

- ¿Qué clase de función H se utilizará?
- ¿Cómo resolver la situación de que H no produzca la posición del elemento asociado?

La respuesta a la segunda cuestión es que se debe utilizar algún método para producir una posición alternativa, es decir, el índice d', y si ésta no es aún la posición del elemento deseado, se produce un tercer índice d'', y así sucesivamente. El caso en el que una clave distinta de la deseada está en la posición identificada se denomina *colisión*; la tarea de generación de índices alternativos se denomina tratamiento de colisiones.

Un ejemplo de colisiones puede ser:

```
clave 345123124
clave 416457234          función de conversión H → dirección 200
                         función de conversión H → dirección 200
```

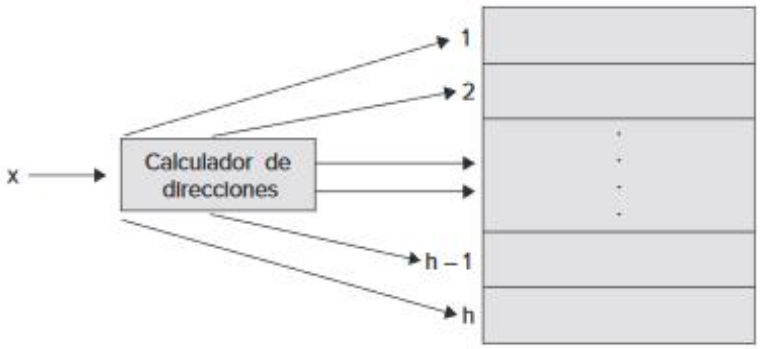
Dos claves distintas producen la misma dirección, es decir, *colisiones*. La elección de una buena función de conversión exige un tratamiento idóneo de colisiones, es decir, la reducción del número de colisiones.

Cristian Pilaparta 1096211-2202112331

10.3.3.1. *Métodos de transformación de claves*

Existen numerosos métodos de transformación de claves.

Todos ellos tienen en común la necesidad de convertir claves en direcciones. En esencia, la función de conversión equivale a una caja negra que podríamos llamar *calculador de direcciones*. Cuando se desea localizar un elemento de clave *x*, el indicador de direcciones indicará en qué posición del array estará situado el elemento.



Truncamiento

Ignora parte de la clave y se utiliza la parte restante directamente como índice (considerando campos no numéricos y sus códigos numéricos). Si las claves, por ejemplo, son enteros de ocho dígitos y la tabla de transformación tiene mil posiciones, entonces el primero, segundo y quinto dígitos desde la derecha pueden formar la función de conversión. Por ejemplo, 72588495 se convierte en 895. El truncamiento es un método muy rápido, pero falla para distribuir las claves de modo uniforme.

Plegamiento

La técnica del plegamiento consiste en la partición de la clave en diferentes partes y la combinación de las partes en un modo conveniente (a menudo utilizando suma o multiplicación) para obtener el índice.

La clave *x* se divide en varias partes, $x_1, x_2, \ldots, x_n$, donde cada parte, con la única posible excepción de la última, tiene el mismo número de dígitos que la dirección más alta que podría ser utilizada.

A continuación se suman todas las partes:

$$h(x) = x_1 + x_2 + \ldots + x_n$$

En esta operación se desprecian los dígitos más significativos que se obtengan de arrastre o acarreo.

EJEMPLO 10.8

Un entero de ocho dígitos se puede dividir en grupos de tres, tres y dos dígitos, los grupos se suman juntos y se truncan si es necesario para que estén en el rango adecuado de índices.

Por consiguiente, si la clave es:

```
62538194
```

y el número de direcciones es 100, la función de conversión será:

```
625 + 381 + 94 = 1100
```

que se truncará a 100 y que será la dirección deseada.

Cristian Pilaparia 1095211-2202112331

EJEMPLO 10.9

Los números empleados –campo clave– de una empresa constan de cuatro dígitos y las direcciones reales son 100. Se desea calcular las direcciones correspondientes por el método de plegamiento de los empleados.

```
4205    8148    3355
```

Solución

```
h(4205) = 42 + 05 = 47
h(8148) = 81 + 48 = 129   y se convierte en 29 (129 – 100), es decir, se ignora el acarreo 1
h(3355) = 33 + 55 = 88
```

Si se desea afinar más se podría hacer la inversa de las partes pares y luego sumarlas.

Aritmética modular

Convertir la clave a un entero, dividir por el tamaño del rango del índice y tomar el resto como resultado. La función de conversión utilizada es **mod** (módulo o resto de la división entera).

```
h(x) = x mod m
```

donde *m* es el tamaño del array con índices de 0 a *m* – 1. Los valores de la función –direcciones– (el resto) irán de 0 a *m* – 1, ligeramente menor que el tamaño del array. La mejor elección de los módulos son los números primos. Por ejemplo, en un array de 1.000 elementos se puede elegir 997 o 1.009. Otros ejemplos son:

```
18 mod 6      19 mod 6      20 mod 6
```

que proporcionan unos restos de 0, 1 y 2, respectivamente.

Si se desea que las direcciones vayan de 0 hasta *m*, la función de conversión debe ser:

```
h(x) = x mod (m + 1)
```

EJEMPLO 10.10

Un vector T tiene cien posiciones, 0... 100. Supongamos que las claves de búsqueda de los elementos de la tabla son enteros positivos (por ejemplo, número del DNI).

Una función de conversión h debe tomar un número arbitrario entero positivo x y convertirlo en un entero en el rango 0... 100, esto es, h es una función tal que para un entero positivo x.

$h(x) = n$, donde n es entero en el rango 0... 100

El método del módulo, tomando 101, será:

```
h(x) = x mod 101
```

Si se tiene el DNI número 234661234, por ejemplo, se tendrá la posición 56:

```
234661234 mod 101 = 56
```

EJEMPLO 10.11

La clave de búsqueda es una cadena de caracteres –tal como un nombre–. Obtener las direcciones de conversión.

El método más simple es asignar a cada carácter de la cadena un valor entero (por ejemplo, A = 1, B = 2, ...) y sumar los valores de los caracteres en la cadena. Al resultado se le aplica entonces el módulo 101, por ejemplo.

Si el nombre fuese JONAS, esta clave se convertiría en el entero:

```
10 + 15 + 14 + 1 + 19 = 63
63 mod 101 = 63
```

Cristian Pilaparta 1095211-2202112331

Mitad del cuadrado

Este método consiste en calcular el cuadrado de la clave x. La función de conversión se define como:

$h(x) = c$

donde c se obtiene eliminando dígitos a ambos extremos de x^2. Se deben utilizar las mismas posiciones de x^2 para todas las claves.

EJEMPLO 10.12

Una empresa tiene ochenta empleados y cada uno tiene un número de identificación de cuatro dígitos y el conjunto de direcciones de memoria varía en el rango de 0 a 100. Calcular las direcciones que se obtendrán al aplicar función de conversión por la mitad del cuadrado de los números empleados:

```
4205    7148    3350
```

Solución

```
x          4205          7148          3350
x²    17 682 025    51 093 904    11 122 250
```

Si elegimos, por ejemplo, el cuarto y quinto dígito significativo, quedaría:

```
h(x)    82    93    22
```

10.3.3.2. *Colisiones*

La función de conversión $h(x)$ no siempre proporciona valores distintos, puede suceder que para dos claves diferentes x_1 y x_2 se obtenga la misma dirección. Esta situación se denomina *colisión* y se deben encontrar métodos para su correcta resolución.

Retomemos los ejemplos vistos anteriormente de las claves DNI correspondientes al archivo de empleados, en el caso de cien posibles direcciones. Si se considera el método del módulo en el caso de las claves, y se considera el número primero 101:

```
123445678    123445880
```

proporcionarían las direcciones:

```
h (123445678) - 123445678 mod 101 - 44
h (123445880) - 123445880 mod 101 - 44
```

Es decir, se tienen dos elementos en la misma posición del vector o array, `[44]`. En terminología de claves se dice que las claves 123445678 y 123445880 han *colisionado*.

El único medio para evitar totalmente el problema de las colisiones es tener una posición del array para cada posible número de DNI. Si, por ejemplo, los números de DNI son las claves y el DNI se representa con nueve dígitos, se necesitaría una posición del array para cada entero en el rango 000000000 a 999999999. Evidentemente, sería necesaria una gran cantidad de almacenamiento. En general, el único método para evitar colisiones totalmente es que el array sea lo bastante grande para que cada posible valor de la clave de búsqueda pueda tener su propia posición. Ya que esto normalmente no es práctico ni posible, se necesitará un medio para tratar o resolver las colisiones cuando sucedan.

Resolución de colisiones

Consideremos el problema producido por una colisión. Supongamos que desea insertar un elemento con número nacional de identidad DNI 12345678, en un array `T`. Se aplica la función de conversión del módulo y se determina

Cristian Pilapaña 1095211-2202112331

que el nuevo elemento se situará en la posición T[44]. Sin embargo, se observa que T[44] ya contiene un elemento con DNI 123445779.

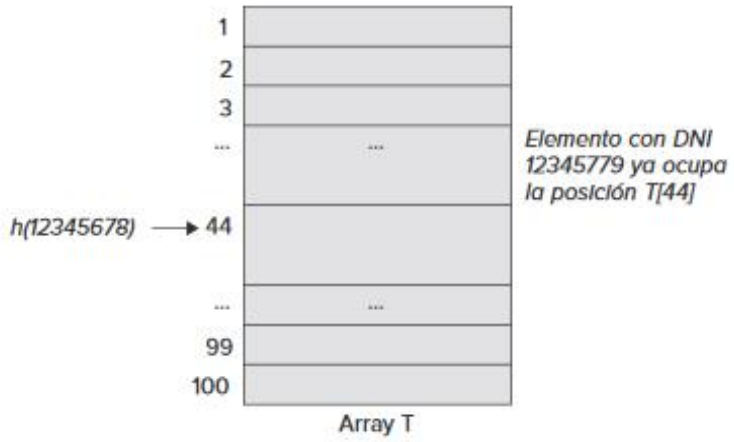


Figura 10.4. Colisión.

La pregunta que se plantea inmediatamente es ¿qué hacer con el nuevo elemento?

Un método comúnmente utilizado para resolver una colisión es cambiar la estructura del array T de modo que pueda alojar más de un elemento en la misma posición. Se puede, por ejemplo, modificar T de manera que cada posición T[i] sea por sí misma un array capaz de contener N elementos. El problema, evidentemente, será saber la magnitud de N. Si N es muy pequeño, el problema de las colisiones aparecerá cuando aparezca N + 1 elementos.

Una solución mejor es permitir una lista enlazada o encadenada de elementos para formar a partir de cada posición del array. En este método de resolución de colisiones, conocido como *encadenamiento*, cada entrada T[i] es un puntero que apunta al elemento del principio de la lista de elementos (capítulo 12), de modo que la función de transformación de clave lo convierte en la posición *i*.

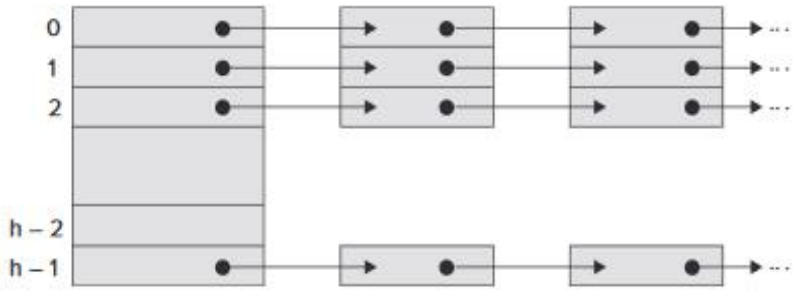


Figura 10.5. Encadenamiento.

10.4. Intercalación

La *intercalación* es el proceso de mezclar (intercalar) dos vectores ordenados y producir un nuevo vector ordenado.

Consideremos los vectores (listas de elementos) ordenados:

```
A:   6   23   34
B:   5   22   26   27   39
```

El vector clasificado es:

```
C:   5   6   22   23   24   26   27   39
```

Cristian Pilaparia 1095211-2202112331

La acción requerida para solucionar el problema es muy fácil de visualizar. Un algoritmo sencillo puede ser:

1. Poner todos los valores del vector A en el vector C.
2. Poner todos los valores del vector B en el vector C.
3. Clasificar el vector C.

Es decir, todos los valores se ponen en el vector C con todos los valores de A, seguidos por todos los valores de B. A continuación se clasifica el vector C. Evidentemente es una solución correcta. Sin embargo, se ignora por completo el hecho de que los vectores A y B están clasificados.

Supongamos que los vectores A y B tienen M y N elementos. El vector C tendrá M + N elementos.

El algoritmo comenzará seleccionando el más pequeño de los dos elementos A y B, situándolo en C. Para poder realizar las comparaciones sucesivas y la creación del nuevo vector C, necesitaremos dos índices para los vectores A y B. Por ejemplo, *i* y *j*. Entonces nos referiremos al elemento *i* en la lista A y al elemento j en la lista B. Los pasos generales del algoritmo son:

```
si elemento i de A es menor que elemento j de B entonces
   transferir elemento i de A a C
   avanzar i (incrementar en 1)
si_no
   transferir elemento j de B a C
   avanzar j
fin_si
```

Se necesita un índice K que represente la posición que se va rellenando en el vector C. El proceso gráfico se muestra en la figura 10.6.

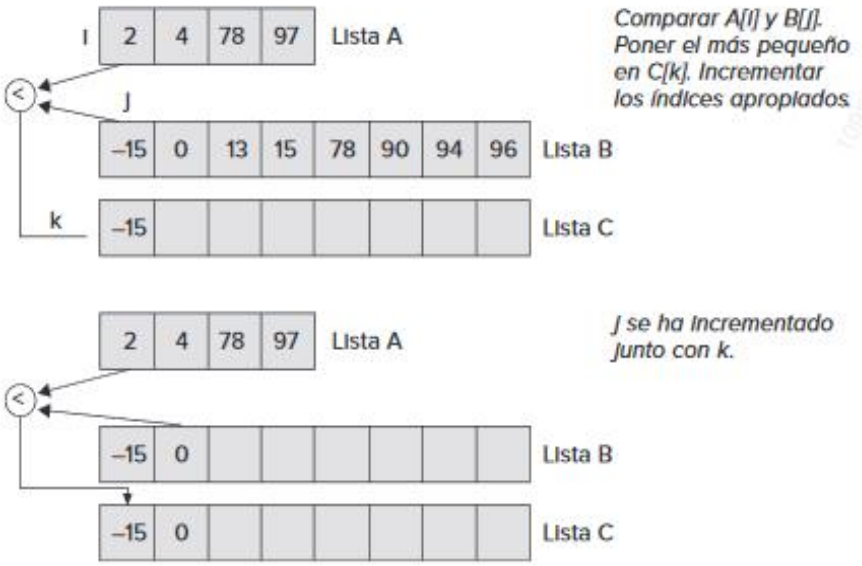


Figura 10.6. Intercalación (B[j] < A[i], de modo que C[k] se obtiene de B[j]).

El primer refinamiento del algoritmo:

```
{estado inicial de los algoritmos}
i ← 1
j ← 1
k ← 0
mientras (i <= M) y (j <= N) hacer
   //seleccionar siguiente elemento de A o B y añadir a C
   k ← k + 1
   //incrementar K}
```

Cristian Pilaparta 1095211-2202112331

```
    si A[i] < B[j] entonces
       C[k] ← A[i]
       i ← i + 1
    si_no
       C[k] ← B[j]
       j ← j + 1
    fin_si
fin_mientras
```

Si los vectores tienen elementos diferentes, el algoritmo anterior no requiere seguir haciendo comparaciones cuando el vector más pequeño se termine de situar en C. La operación siguiente deberá copiar en C los elementos que restan del vector más grande. Así, por ejemplo, supongamos:

```
A  =  6      23    24                  i  =  4
B  =  5      22    26    27    39      j  =  3
C  =  5       6    22    23    24      k  =  5
```

Todos los elementos del vector A se han relacionado y situado en el vector C. El vector B contiene los elementos no seleccionados y que deben ser copiados, en orden, al final del vector C. En general, será necesario decidir cuál de los vectores A o B tienen elementos no seleccionados y a continuación ejecutar la asignación necesaria.

El algoritmo de copia de los elementos restantes es:

```
si i <= M entonces
   desde r ← i hasta M hacer
      k ← k + 1
      C[k] ← A[r]
   fin_desde
si_no
   desde r ← j hasta N hacer
      k ← k + 1
      C[k] ← B[r]
   fin_desde
fin_si
```

El algoritmo total resultante de la intercalación de dos vectores A y B ordenados en uno C es:

```
algoritmo intercalación
inicio
   leer(A, B)   //A, B vectores de M y N elementos
   i ← 1
   j ← 1
   k ← 0

   mientras (i <= M) y (j <= N) hacer
      //seleccionar siguiente elemento de A o B y añadirlo a C
      k ← k+1
      si A[i] < B[j] entonces
         C[k] ← A[i]
         i ← i + 1
      si_no
         C[k] ← B[j]
         j ← j + 1
      fin_si
   fin_mientras
   //copiar el vector restante
   si i <= M entonces
```

Cristian Pilapanta 1095211-2202112331

```
      desde r ← i hasta M hacer
         k ← k + 1
         C[k] ← A[r]
      fin_desde
   si_no
      desde r ← j hasta N hacer
         k ← k + 1
         C[k] ← B[r]
      fin_desde
   fin_si
   escribir(C)   //vector clasificado
fin
```

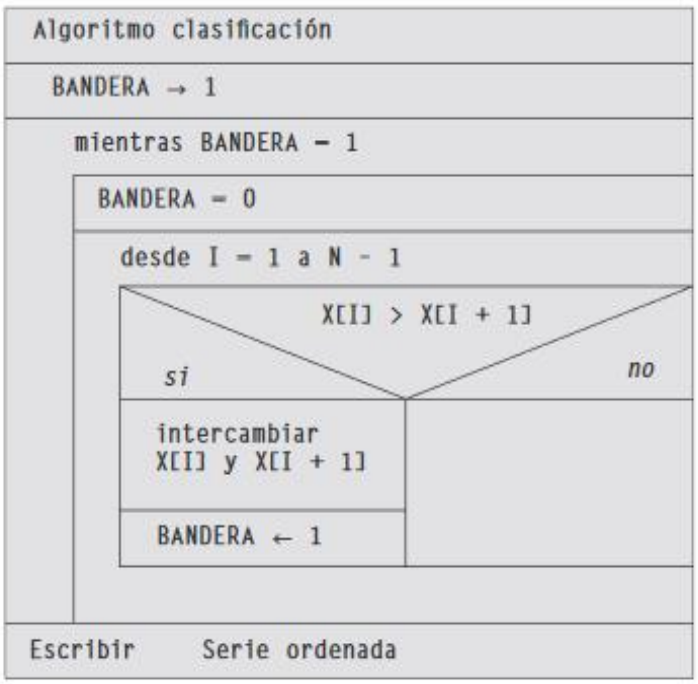
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

10.1. *Clasificar una serie de números* X1, X2, ..., Xn *en orden creciente por el método del intercambio o de la burbuja.*

Análisis

Se utiliza un indicador (bandera) igual a 0 si la serie está bien ordenada y a 1 en caso contrario. Como *a priori* la serie no está bien ordenada, se inicializa el valor de la bandera a 1 y después se repiten las siguientes acciones:

- Se fija la bandera a 0.
- A partir del primero se comparan dos elementos consecutivos de la serie; si están bien ordenados se pasa al elemento siguiente, si no se intercambian los valores de los dos elementos y se fija el valor de la bandera a 1; si después de haber pasado revista —leído— toda la serie, la bandera permanece igual a 0, entonces la clasificación está terminada.



Cristian Pilaparra 1095211-2202112331

10.2. *Clasificar los números* A *y* B.

Método 1

```
algoritmo clasificar
inicio
   leer(A, B)
   si A < B entonces
      permutar (A , B)
   fin_si
   escribir('Más grande', A)
   escribir('Más pequeña', B)
fin
```

Método 2

```
algoritmo clasificar
inicio
   leer(A)
   MAX ← A
   leer(B)
   MIN ← B
   si B > A entonces
      MAX ← B
      MIN ← A
   fin_si
   escribir('Máximo —', MAX)
   escribir('Mínimo —', MÍN)
fin
```

10.3. *Se dispone de una lista de números enteros clasificados en orden creciente. Se desea conocer si un número dado introducido desde el terminal se encuentra en la lista. En caso afirmativo, averiguar su posición, y en caso negativo, se desea conocer su posición en la lista e insertarlo en su posición.*

Análisis

Como ya conoce el lector, existen dos métodos fundamentales de búsqueda: lineal y binaria. Resolvemos el problema con los dos métodos a fin de consolidar las ideas sobre ambos.

Búsqueda lineal

El método consiste en comparar el número dado en orden sucesivo con todos los elementos del conjunto de números, efectuando un recorrido completo del vector que representa la lista.

El proceso termina cuando se encuentra un número igual o superior al número dado.

El método de inserción o intercalación de un elemento en el vector será el descrito en el apartado 6.3.4.

La tabla de variables es la siguiente:

N	número de elementos de la lista: *entero*.
J	posición del elemento en la lista: *entero*.
K	contador del bucle de búsqueda: *entero*.
X	número dado: *entero*.
LISTA	conjunto de números enteros.

Búsqueda dicotómica

La condición para realizar este método —más rápido y eficaz— es que la lista debe estar clasificada en orden creciente o decreciente.

Se obtiene el número de elementos de la lista y se calcula el número central de la lista.

Si el número dado es igual al número central de la lista, la búsqueda ha terminado; en caso contrario, pueden suceder dos casos:

Cristian Pilaquinga 1095211-2202112331

- El número está en la sublista inferior.
- El número está en la sublista superior.

Tras localizar la sublista donde se encuentra, se consideran variables MÍN y MÁX que contienen los elementos menor y mayor de cada sublista —que coincidirán con los extremos al estar ordenada la lista—, así como el término central (CENTRAL), de acuerdo con el siguiente esquema.

Sublista inferior	L[1] L[2] ...L[CENTRAL]
Sublista superior	L[CENTRAL + 1]...L[N]

Los valores de las variables INF, SUP y CENTRAL serán:

Primera búsqueda

$$\text{CENTRAL} = \frac{(\text{SUP} - \text{INF})}{2} + \frac{\text{INF} - \text{N} - 1}{2} + 1 = \frac{\text{N} - 1}{2}$$

```
SUP = N
INF = 1
```

- *Si el número X está en la sublista inferior, entonces*

```
INF = 1
SUP = CENTRAL - 1
```

y se realiza una segunda búsqueda entre los elementos de orden 1 y CENTRAL.

- *Si el número X está en la sublista superior, entonces*

```
INF = CENTRAL + 1
SUP = N
```

y se realiza una segunda búsqueda entre los elementos de orden CENTRAL + 1 y N.

El proceso de variables es:

N	número de elementos de la lista: *entero*.
I	contador del bucle de búsqueda: *entero*.
SW	interruptor o bandera para indicar si el número dado está en la lista: *lógico*.
LISTA	conjunto de números enteros: *entero*.
X	número buscado: *entero*.
INF	posición inicial de la lista o sublista: *entero*.
SUP	posición superior de la lista o sublista: *entero*.
POSICIÓN	lugar del orden ocupado por el número buscado: *entero*.

Pseudocódigo

Búsqueda lineal

```
algoritmo búsqueda_1
var
   entero : I, K, X, N
   array[1..50] de entero : lista
   //se supone dimensión de la lista a 50 elementos y que se trata de una
   //lista ordenada
inicio
   leer(N)
   //lectura de la lista
   desde I ← 1 hasta N hacer
      leer(LISTA[I])
   fin_desde
```

Cristian Pilaparta 1095211-2202112331

```
   Ordenar (LISTA, N)
   leer(X)
   I ← 0
   repetir
      I ← I + 1
   hasta_que (LISTA[I] >= X) o (I = 50)
   si LISTA[I] = X entonces
      escribir('se encuentra en',I)
   si_no
      escribir('El número dado no está en la lista')
      //insertar el elemento X en la lista
      si N < 50 entonces
         desde K ← N hasta I decremento 1 hacer
            LISTA[K + 1] ← LISTA[K]
         fin_desde
         LISTA[I] ← X
         N ← N + 1
         escribir('Insertado en',I)
      fin_si
   fin_si
   //escritura del vector LISTA
   desde I ← 1 hasta N hacer
      escribir(LISTA[I])
   fin_desde
fin
```

Búsqueda dicotómica

```
algoritmo búsqueda_b
var
   entero: I, N, X, K, INF, SUP, CENTRAL, POSICIÓN
   lógico: SW
   array [1...50] de entero: LISTA
inicio
   leer(N)
   desde I ← 1 hasta N hacer
      leer(LISTA[I])   //la lista ha de estar ordenada
   fin_desde
   Ordenar (LISTA, N)
   leer(X)
   SW ← falso
   INF ← 1
   SUP ← N
   repetir
      CENTRAL ← (SUP - INF)DIV 2 + INF
      si LISTA[CENTRAL] = X entonces
         escribir('Número encontrado en la lista')
         POSICIÓN ← CENTRAL
         escribir(POSICIÓN)
         SW ← verdad
      si_no
         si X < LISTA[CENTRAL] entonces
            SUP ← CENTRAL
         si_no
```

Cristian Pilaparita 1095211-2202112331

```
            INF ← CENTRAL+1
         fin_si
         si (INF = SUP) y (LISTA[INF] = X) entonces
            escribir('El número está en la lista')
            POSICIÓN ← INF
            escribir(POSICIÓN)
            SW ← verdad
      fin_si
   fin_si
   hasta_que (INF = SUP) o SW
   si no (SW) entonces
      escribir('Número no existe en la lista')
      si X < lista(INF) entonces
         POSICIÓN ← INF
      si_no
         POSICIÓN ← INF+1
      fin_si
      escribir(POSICIÓN)
      desde K ← N hasta POSICIÓN decremento 1 hacer
         LISTA[K + 1] ← LISTA[K]
      fin_desde
      LISTA[POSICIÓN] ← X
      N ← N + 1
   fin_si
   //escritura de la lista
   desde I ← 1 hasta N hacer
      escribir(LISTA[I])
   fin_desde
fin
```

10.4. *Ordenar de mayor a menor un vector de N elementos (N <= 40), cada uno de los cuales es un registro con los campos día, mes y año de tipo entero.*

Utilice una función ESMENOR(fecha1,fecha2) que nos devuelva si una fecha es menor que otra.

```
algoritmo ordfechas
tipo registro: fechas
      inicio
         entero: día
         entero: mes
         entero: año
      fin_registro
      array[1..40] de fechas: arr
var arr    : f
   entero  : n
inicio
   pedirfechas(f,n)
   ordenarfechas(f,n)
   presentarfechas(f,n)
fin
lógico función esmenor(E fechas: fecha1,fecha2)
inicio
   si (fecha1.año<fecha2.año) o
      (fecha1.año=fecha2.año) y (fecha1.mes<fecha2.mes) o
```

Cristian Pilapanta 1095211-2202112331

```
      (fecha1.año=fecha2.año) y (fecha1.mes=fecha2.mes) y
      (fecha1.día<fecha2.día) entonces
         devolver(verdad)
   si_no
         devolver(falso)
   fin_si
fin_función

procedimiento pedirfechas(S arr:f; S entero:n)
var entero:i
    entero:día
inicio
   i←1
   escribir ("Deme la",i,"ª fecha")
   escribir("Día:")
   leer(día)
   mientras (día<>0) y (i<=40) hacer
      f[i].día ← día
      escribir("Mes:")
      leer(f[i].mes)
      escribir("Año:")
      leer(f[i].año)
      n ← i
      i ← i+1
      si i<=40 entonces
         escribir ("Deme la ",i,"ª fecha")
         escribir ("Día: ")
         leer(día)
      fin_si
   fin_mientras
fin_procedimiento

procedimiento ordenarfechas(E/S arr:f; E entero:n)
   var entero:salto
      lógico :ordenada
      entero :j
      fechas :AUXI
inicio
   salto ← n
   mientras salto > 1 hacer
      salto ← salto div 2
      repetir
      ordenada ← verdad
      desde j ← 1 hasta n-salto hacer
      si esmenor(f[j], f[j+salto]) entonces
         AUXI ← f[j]
         f[j] ← f[j+salto]
         f[j+salto] ← AUXI
         ordenada ← falso
      fin_si
      fin_desde
      hasta ordenada
   fin_mientras
fin_procedimiento

procedimiento presentarfechas(E arr:f; E entero:n)
   var entero:i
```

Cristian Pilaparda 1095211-2202112331

```
   inicio
   desde i←1 hasta n hacer
      escribir(f[i].día,f[i].mes,f[i].año)
   fin_desde
fin_procedimiento
```

Considere otras posibilidades, usando el mismo método de ordenación, para resolver el ejercicio.

10.5. *Dada la lista de fechas ordenada en orden decreciente del ejercicio anterior, diseñar los procedimientos:*

1. Buscar, que nos informará sobre si una determinada fecha se encuentra o no en la lista:
   - Si no está, indicará la posición donde correspondería insertarla.
   - Si está, nos dirá la posición donde la hemos encontrado o, si estuviera repetida, a partir de qué posición y cuántas veces son las que aparece.
2. Insertar, que nos permitirá insertar una fecha en una determinada posición. Se deberá utilizar en un algoritmo haciendo uso previo de buscar; así, cuando una fecha no se encuentre en la lista, la insertará en el lugar adecuado para que no se pierda la ordenación inicial.

```
algoritmo buscar_insertar_fechas
tipo registro: fechas
      inicio
         entero: día
         entero: mes
         entero: año
      fin_registro
      array[1..40] de fechas: vector

var vector : f
   entero  : n
   fechas  : fecha
   lógico  : esta
   entero  : posic, cont

inicio
   pedirfechas(f,n)
   ordenarfechas(f,n)
   presentarfechas(f,n)
   escribir('Deme fecha a buscar (dd mm aa)')
   leer(fecha.día,fecha.mes,fecha.año)
   buscar(f,n,fecha,esta,posic,cont)
   si esta entonces
      si cont > 1 entonces
         escribir('Aparece a partir de la posición: ', posic, ' ', cont, ' veces')
      si_no
         escribir('Está en la posición: ', posic )
      fin_si
   si_no
      si n=40 entonces
         escribir('No está. Array lleno')
      si_no
         insertar(f,n,fecha,posic)
         presentarfechas(f,n)
      fin_si
fin
```

Cristian Pilaparia 1095211-2202112331

```
lógico función esmenor(E fechas: fecha1,fecha2)
   inicio
      ......................
   fin_función

lógico función esigual(E fechas: fecha1,fecha2)
inicio
   si (fecha1.año=fecha2.año) y (fecha1.mes=fecha2.mes) y (fecha1.día=fecha2.día) entonces
      devolver(verdad)
   si_no
      devolver(falso)
   fin_si
fin_función

procedimiento pedirfechas(S vector: f; S entero n)
   var entero: i
      entero : día
   inicio
      ...
fin_procedimiento

procedimiento ordenarfechas(E/S vector: f; E entero: n)
   var entero : salto
      lógico  : ordenada
      entero  : j
      fechas  : AUXIi
   inicio
   ...
fin_procedimiento

procedimiento buscar(E vector: f; E entero:n; E fechas: fecha; S lógico:está; S entero:
                     posic, cont)
var entero : primero,último,central,i
   lógico : encontrado
inicio
   primero ← 1
   último ← n
   esta ← falso
   mientras (primero<=último) y (no está) hacer
      central ← (primero+último) div2
      si esigual(f[central],fecha) entonces
            está ← verdad
      si_no
         si esmenor(f[central],fecha) entonces
            último ← central-1
         si_no
            primero ← central+1
         fin_si
      fin_si

   fin_mientras
   cont ← 0
   si esta entonces
```

Cristian Pilapanta 1095211-2202112331

```
        i ← central-1
        encontrado ← verdad
        mientras (i>=1) y (encontrado) hacer
            si esigual(f[i],f[central]) entonces
               i ← i-1
            si_no
               encontrado ← falso
            fin_si
         fin_mientras
         i ← i+1
         encontrado ← verdad
         posic ← i
         mientras (i<=40) y encontrado hacer
            si esigual(f[i],f[central]) entonces
               cont ← cont+1
                  i ← i+1
            si_no
               encontrado ← falso
            fin_si
         fin_mientras
      si_no
         posic ← primero
   fin_si
fin_procedimiento

procedimiento insertar(E/S vector: f; E/S entero: n
E fechas:fecha; E entero:posic)
var entero:i
inicio
   desde i ← n hasta posic decremento 1 hacer
      f[i+1] ← f[i]
   fin_desde
   f[posic] ← fecha
   n ← n+1
   fin_procedimiento

procedimiento presentarfechas(E vector:f; E entero:n)
   var entero:i
   inicio
      ...
   fin_procedimiento
```

10.6. *Escriba el procedimiento de búsqueda binaria de forma recursiva.*

```
algoritmo búsqueda_binaria
   tipo
      array[1..10] de entero: arr
   var
      arr : a
      entero : núm, posic, i
inicio
   desde i ← 1 hasta 10 hacer
```

Cristian Pilaparta 1095211-2202112331

```
      leer(a[i])
   fin_desde
   ordenar(a)
   escribir('Indique el número a buscar en el array ')
   leer(núm)
   búsqueda(a,posic, 1, 10, núm)
   si posic > 0 entonces
      escribir('Existe el elemento en la posición ', posic)
   si_no
      escribir('No existe el elemento en el array.')
   fin_si
fin

procedimiento ordenar(E/S arr: a)
   ...
   inicio
      ...
   fin_procedimiento

procedimiento búsqueda(E arr: a; S entero: posic
                       E entero: primero,último,núm)
   //Este procedimiento devuelve 0 si no existe el elemento
   en el array, y si existe devuelve su posición
   var
      entero: central
   inicio
      si primero > último entonces
         posic ← 0
      si_no
         central ← (primero+último) div 2
            si a[central] = núm entonces
               posic ← central
      si_no
         si núm > a[central] entonces
            primero ← central + 1
         si_no
            último ← central - 1
         fin_si
         búsqueda(a, posic, primero, último, núm)
      fin_si
   fin_si
fin_procedimiento
```

10.7. *Partiendo de la siguiente lista inicial:*

80 36 98 62 26 78 22 27 2 45

tome como elemento pivote el contenido del que ocupa la posición central y realice el seguimiento de los distintos pasos que llevarían a su ordenación por el método *Quick-Sort*. Implemente el algoritmo correspondiente.

Cristian Pilaparita 1095211-2202112331

1	2	3	4	5	6	7	8	9	10
80	**36**	**98**	**62**	**26**	**78**	**22**	**27**	**2**	**45**
2								**80**	
	22					**36**			
		26		**98**					
		j	i						

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**62**	**98**	**78**	**36**	**27**	**80**	**45**
	22								
j		i							

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**62**	**98**	**78**	**36**	**27**	**80**	**45**
			27				**62**		
				36		**98**			
				j	i				

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**78**	**98**	**62**	**80**	**45**
			27						
		j		i					

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**78**	**98**	**62**	**80**	**45**
					45				**78**
						62	**98**		
						j	i		

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**45**	**62**	**98**	**80**	**78**
					45				
				j		i			

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**45**	**62**	**98**	**80**	**78**
							78		**98**
								80	
							j		i

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**45**	**62**	**78**	**80**	**98**

```
algoritmo quicksort
tipo
array[1..10] de entero: arr
var
   arr    : a
   entero : k
```

Cristian Pilaparita 1095211-2202112331

```
inicio
   desde k ← 1 hasta 10 hacer
      leer (a[k])
   fin_desde
   rápido (a,10)
   desde k ← 1 hasta 10 hacer
      escribir (a[k])
   fin_desde
fin

   procedimiento intercambiar (E/S entero: m,n)
      var
         entero: AUXI
      inicio
         AUXI ← m
         m ← n
         n ← AUXI
   fin_procedimiento

   procedimiento partir (E/S arr: a E entero: primero, último)
      var
         entero: i,j,central
   inicio
      i ← primero
      j ← último
      // encontrar elemento pivote, central, y almacenar su contenido
      central ← a[ (primero+último) div 2 ]
      repetir
         mientras a[i] < central hacer
            i ← i+1
         fin_mientras
         mientras a[j] > central hacer
            j ← j-1
         fin_mientras
         si i <= j entonces
            intercambiar( a[i],a[j] )
            i ← i+1
            j ← j-1
         fin_si
      hasta_que i > j
      si primero < j entonces
         partir (a,primero,j)
      fin_si
      si i < último entonces
         partir (a,i,último)
      fin_si
   fin_procedimiento

   procedimiento rápido (E/S arr: a; E entero: n)
      inicio
         partir (a,1,n)
      fin_procedimiento
```

Cristian Pilaparda 1095211-2202112332
1095211220211233

CONCEPTOS CLAVE

- Búsqueda
- Eficiencia de los métodos de ordenación
- Intercalación
- Ordenación
- Tipos de búsqueda

RESUMEN

La ordenación de datos es una de las aplicaciones más importantes de las computadoras. Dado que es frecuente que un programa trabaje con grandes cantidades de datos almacenados en arrays, resulta imprescindible conocer diversos métodos de ordenación de arrays y cómo, además, puede ser necesario determinar si un array contiene un valor que coincide con un cierto valor clave; también resulta básico conocer los algoritmos de búsqueda.

1. La *ordenación* o *clasificación* es el proceso de organizar datos en algún orden o secuencia específica, tal como creciente o decreciente para datos numéricos o alfabéticamente para datos de caracteres. La ordenación de arrays (arreglos) se denomina ordenación interna, ya que se efectúa con todos los datos en la memoria interna de la computadora.
2. Es posible ordenar arrays por diversas técnicas, como burbuja, selección, inserción, *Shell* o *QuickSort* y, cuando el número de elementos a ordenar es pequeño, estos métodos son aceptables.
3. Para ordenar arrays con un gran número de elementos debe tenerse en cuenta la diferente eficiencia en cuanto al tiempo de ejecución entre los métodos comentados. Entre los citados, *QuickSort* y *Shell* son los más avanzados.
4. El método de búsqueda lineal de un determinado valor clave en un array, que compara cada elemento con la clave buscada, puede ser útil en arrays pequeños o no ordenados.
5. El método de búsqueda binaria es mucho más eficiente, pero requiere arrays ordenados.
6. Puesto que los arrays permiten el acceso directo a un determinado elemento o posición, la información en un array no tiene por qué ser colocada en forma secuencial. Es posible, por tanto, usar una función *hash* que transforme el valor clave en un número válido para ser utilizado como subíndice en el array y almacenar la información en la posición especificada por dicho subíndice.
7. Una función de conversión *hash* no siempre proporciona valores distintos, y puede suceder que para dos claves diferentes devuelva la misma dirección. Esta situación se denomina *colisión* y se deben encontrar métodos para su correcta resolución.
8. Entre los métodos para resolver las colisiones destacan:
   a) Reservar una zona especial en el array para colocar las colisiones.
   b) Buscar la primera posición libre que siga a aquella donde se debiese colocar la información y en la que no se pudo situar por encontrarse ocupada debido a la colisión.
   c) Utilizar encadenamiento.
9. Si la información se coloca en un array aplicando una función *hash* a determinado campo clave y estableciendo un método de resolución de colisiones, la consulta por dicho campo clave también se efectuará de forma análoga.
10. Cuando se tienen dos vectores ordenados y se necesita obtener otro también ordenado, el proceso de intercalación o mezcla debe producirnos el resultado deseado, sin que sea necesario aplicar a continuación ningún método de ordenación.

Cristian Pilapaña 1095211-2202112332

EJERCICIOS

10.1. Realizar el diagrama de flujo y el pseudocódigo que permuta tres enteros: *n*1, *n*2 y *n*3 en orden creciente.

10.2. Escribir un algoritmo que lea diez nombres y los ponga en orden alfabético utilizando el método de selección. Utilice los siguientes datos para comprobación: Sánchez, Waterloo, McDonald, Bartolomé, Jorba, Clara, David, Robinson, Francisco, Westfalia.

10.3. Clasificar el array (vector):

```
42  57  14  40  96  19  08  68
```

por los métodos: 1) selección, 2) burbuja. Cada vez que se reorganice el vector, se debe mostrar el nuevo vector reformado.

10.4. Supongamos que se tiene una secuencia de *n* números que deben ser clasificados:

1. Utilizando el método de selección, cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si:
   - Ya está clasificado.
   - Está en orden inverso.
2. Repetir el paso *i* para el método de selección.

10.5. Escribir un algoritmo de búsqueda lineal para un vector ordenado.

10.6. Un algoritmo ha sido diseñado para leer una lista de no más de 1 000 enteros positivos, cada uno menos de 100, y ejecutar algunas operaciones. El cero es la marca final de la lista. El programador debe obtener lo siguiente en el algoritmo:

1. Visualizar los números de la lista en orden creciente.
2. Calcular e imprimir la mediana (valor central).
3. Determinar el número que ocurre más frecuentemente.
4. Imprimir una lista que contenga:
   - Números menores de 30.
   - Números mayores de 70.
   - Números que no pertenezcan a los dos grupos anteriores.
5. Encontrar e imprimir el entero más grande de la lista junto con su posición en la lista antes de que los números hayan sido ordenados.

10.7. Diseñar diferentes algoritmos para insertar un nuevo valor en una lista (vector). La lista debe estar ordenada en orden ascendente antes y después de la inserción.

Cristian Pilapanta 1095211-2202112332

CAPÍTULO 11

Ordenación, búsqueda y fusión externa (archivos)

11.1. Introducción
11.2. Archivos ordenados
11.3. Fusión de archivos
11.4. Partición de archivos
11.5. Clasificación de archivos

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Cristian Pitaparta 1095211-2202112332

1095211220211233

INTRODUCCIÓN

Los sistemas de procesamiento de la información procesan normalmente gran cantidad de información. En estos casos los datos se almacenan sobre soportes de almacenamiento masivo (cintas y discos magnéticos). Los algoritmos de ordenación presentados en el capítulo 10 no son aplicables si la masa de datos no cabe en la memoria central de la computadora y se encuentran almacenados en su soporte como una cinta. En estos casos se suelen colocar en memoria central las fichas que se procesan y a las que se pueda acceder directamente. Normalmente estas técnicas no son muy eficaces y se utilizan técnicas distintas de ordenación. La más importante es la *fusión* o *mezcla*.

Este capítulo realiza una introducción a las técnicas de ordenación, búsqueda y mezcla o fusión externas.

11.1. Introducción

Cuando la masa de datos a procesar es grande y no cabe en la memoria central, los datos se organizan en archivos que, a su vez, se almacenan en dispositivos externos de memoria auxiliar (discos, cintas magnéticas, etcétera).

Las operaciones básicas estudiadas en el capítulo 10, ordenación, búsqueda e intercalación o mezcla, sufren un cambio importante en su concepción, derivado esencialmente del hecho físico de que los datos a procesar no caben en la memoria principal de la computadora.

11.2. Archivos ordenados

El tratamiento de los archivos secuenciales exige que éstos se encuentren ordenados respecto a un campo del registro, denominado *campo clave*.

Supongamos un archivo del personal de una empresa, cuya estructura de registros es la siguiente:

NOMBRE	tipo cadena	(nombre del empleado)
DIRECCIÓN	tipo cadena	(dirección)
FECHA	tipo cadena	(fecha de nacimiento)
SALARIO	tipo numérico	(salario)
CATEGORÍA	tipo cadena	(categoría laboral)
DNI	tipo cadena	(número de DNI)

La clasificación en orden ascendente o descendente se puede realizar con respecto a una clave (nombre, dirección, entre otros); sin embargo, puede ser interesante tener clasificado un fichero por categoría laboral y a su vez se puede tener por cada categoría laboral los registros agrupados por nombres o direcciones. Ello nos lleva a la conclusión de que un archivo puede estar ordenado por un campo clave o una jerarquía de campos.

Se dice que un archivo (estructura del registro: campos *C1*, *C2*, ... C_n) está ordenado principalmente por el campo *C1*, en orden secundario 1 por el campo *C2*, en orden secundario 2 por el campo *C3*, etcétera, en orden secundario *n* por el campo *Cn*. Si el archivo tiene la siguiente organización:

- Los registros aparecen en el archivo según el orden de los valores del campo clave *C1*.
- Si se considera un mismo valor *C1*, los registros aparecen en el orden de los valores del campo *C2*.
- Para un mismo valor de $C(C_i)$ los registros aparecen según el orden de los valores del campo $C_i + 1$, siendo $1 =< i <= n$.

Si se desea ordenar un archivo principalmente por C_1, y en orden secundario 1 por C_2, se necesita:

- Ordenar primero por C_2.
- Ejecutar a continuación una ordenación estable por el campo C_1.

La mayoría de los sistemas operativos actuales disponen de programas estándar (utilidad) que realizan la clasificación de uno o varios archivos (*sort*). En algunos sistemas operativos existe la orden SORT, que permite realizar la clasificación de archivos según ciertos criterios específicos.

Los algoritmos de clasificación externa son muy numerosos y a ellos dedicaremos gran parte de este capítulo.

11.3. Fusión de archivos

La *fusión* o *mezcla de archivos (merge)* consiste en reunir en un archivo los registros de dos o más archivos ordenados por un campo clave T. El resultado será un archivo ordenado por el campo clave T.

Supongamos que se dispone de dos archivos ordenados sobre dos discos de estado sólido (SSD) y que se desea mezclarlos o fundirlos en un solo archivo ordenado. Sean los archivos F1 y F2 almacenados en dos discos diferentes. El archivo F3 se construye en un tercer disco.

Cristian Pitaparra 1095211-2202112332

El algoritmo de fusión de archivos será:

```
inicio
   //fusión de dos archivos
   1. poner archivo 1 en disco 1, archivo 2 en disco 2
   2. seleccionar de los dos primeros registros de archivo
      1 y archivo 2 el registro de clave más pequeña y almacenarlo en un nuevo archivo 3
   3. mientras (archivo 1 no vacio) y (archivo 2 no vacío) hacer
   4. seleccionar el registro siguiente con clave más
      pequeña y almacenarlo en el archivo 3
      fin_mientras
      //uno de los archivos no está aún vacío
   5. almacenar resto archivo en archivo 3 registro a registro
fin
```

EJEMPLO 11.1

Se dispone de dos archivos, F1 *y* F2, *cuyos campos claves son:*

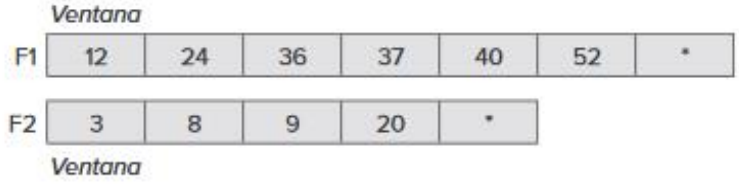
```
F1   12   24   36   37   40   52
F2    3    8    9   20
```

y se desea un archivo FR ordenado, que contenga los dos archivos F1 y F2.

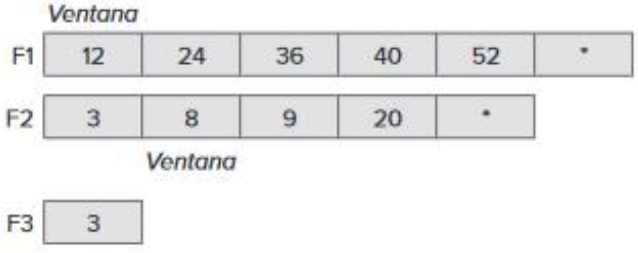
La estructura de los archivos F1 y F2 es:

```
F1   12   24   36   37   40   52   EOF(*) Fin archivo (eof)
F2    3    8    9   20
```

Para realizar la fusión de F1 y F2 es preciso acceder a los archivos F1 y F2 que se encuentran en soportes magnéticos en organización secuencial. En cada operación de acceso a un archivo sólo se puede acceder a un único elemento del archivo en un momento dado. Para realizar la operación se utiliza una variable de trabajo del mismo tipo que los elementos del archivo. Esta variable representa al elemento actual del archivo y denominaremos *ventana*, debido a que será la variable que nos permitirá *ver* el archivo, elemento tras elemento. El archivo se recorre en un único sentido y su final físico termina con una marca especial denominada *fin de archivo (EOF, end of file)*; por ejemplo, un asterisco (*).



Se comparan las claves de las ventanas y se sitúa la más pequeña 3(F2) en el archivo de salida. A continuación, se avanza un elemento el archivo F2 y se realiza una nueva comparación de los elementos situados en las ventanas.



Cristian Pilaparita 1095211-2202112332

Cuando uno u otro de los archivos de entrada se ha terminado, se copia el resto sobre el archivo de salida y el resultado final será:

FR	3	8	9	12	20	24	36	37	40	52	*

El algoritmo correspondiente de fusión de archivos será:

```
algoritmo fusión_archivo
var
   entero:ventana1, ventana2, ventanaS
   archivo_s de entero: F1,F2,F3
   //ventana1,ventana2   claves de los archivos F1,F2
      ventanaS           claves del archivo FR
inicio
   abrir (F1, l, 'nombre')
   abrir (F2, l, 'nombre2')
   crear (FR, 'nombre3')
   abrir (FR, e, 'nombre3')
   leer (F1, ventana1)
   leer (F2, ventana2)
   mientras no FDA(F1) y no FDA (F2) hacer
      si ventana1 <= ventana2 entonces
         ventanaS ← ventana1
         escribir(FR, ventanaS)
         leer (F1, ventana1)
      si_no
         ventanaS ← ventana2
         escribir (FR, ventanaS)
         leer (F2, ventana2)
      fin_si
   fin_mientras
   //lectura terminada de F1 o F2
   mientras no FDA (F1) hacer
      ventanas ← ventana1
      escribir(FR, ventanaS)
      leer(F1, ventana1)
   fin_mientras
   mientras_no FDA(F2)hacer
      ventanaS ← ventana2
      escribir (FR, ventanaS)
      leer(F2, ventana2)
   fin_mientras
   cerrar (F1,F2,FR)
fin
```

Se considera ahora el otro caso posible en los archivos secuenciales. El final físico del archivo se detecta al leer el último elemento (no la marca de fin de archivo) y los ficheros son de registros con varios campos. El algoritmo correspondiente a la *fusión* sería:

```
tipo
   registro: datos_personales
      -: C              //campo por el que están ordenados
      -:-
   fin_registro
   archivo_s de datos_personales: arch
```

Cristian Pilaparta 1095211-2202112332

```
var
   datos_personales:r1, r2
   arch: f1, f2, f       //f es el fichero resultante
   lógico: fin1, fin2
inicio
   abrir (f1, l, 'nombre1')
   abrir (f2, l, 'nombre2')
   crear (f, 'nombre3')
   abrir (f, e, 'nombre3')
   fin1← falso
   fin2← falso
   si FDA (f1) entonces
      fin1← verdad
   si-no
      leer_reg (f1, r1)
fin_si
si FDA (f2) entonces
   fin2← verdad
si_no
   leer_reg (f2, r2)
fin_si
mientras NO fin1 y NO fin2 hacer
   si r1.c <      r2.c entonces
      escribir_reg (f, r1)
      si FDA (f1) entonces
         fin1← verdad
      si_no
         leer_reg (f1, r1)
      fin_si
   si_no
      escribir_reg (f, r2)
      si FDA (f2) entonces
         fin2← verdad
      si_no
         leer_reg (f2, r2)
      fin_si
   fin_si
   fin_mientras
   mientras NO fin1 hacer
      escribir_reg (f, r1)
      si FDA (f1) entonces
         fin1← verdad
      si_no
         leer_reg (f1, r1)
      fin_si
   fin_mientras
   mientras NO fin2 hacer
      escribir_reg (f, r2)
      si FDA (f2) entonces
         fin2← verdad
      si_no
         leer_reg (f2, r2)
```

Cristian Pilapanta 1095211-2202112332
1095211220211233

```
        fin_si
    fin_mientras
    cerrar (f1, f2, f)
fin
```

11.4. Partición de archivos

La *partición* o *división* consiste en repartir los registros de un archivo en otros dos o más en función de una determinada condición.

Aunque existen muchos métodos de producir particiones a partir de un archivo no clasificado, consideraremos sólo los siguientes métodos:

- *Clasificación interna.*
- *Por el contenido.*
- *Selección por sustitución.*
- *Secuencias.*

Supongamos el archivo de entrada siguiente, en el que se indican las claves de los registros:

110	48	33	69	46	2	62	39	28	47	16	19	34	55
99	78	75	40	35	87	10	26	61	92	99	75	11	2
28	16	80	73	18	12	89	50	47	36	67	94	23	15
84	44	53	60	10	39	76	18	24	86				

11.4.1. Clasificación interna

El método más sencillo consiste en leer M registros a la vez de un archivo no clasificado, clasificarlos utilizando un método de clasificación interna y a continuación darles salida como partición. Observe que todas las particiones producidas de este modo, excepto posiblemente la última, contendrán exactamente M registros. La figura muestra las particiones producidas a partir del archivo de entrada de la figura utilizada un tamaño de memoria (M) de cinco registros.

33	46	48	69	110
2	28	39	47	62
16	19	34	55	99
35	40	75	78	87
10	26	61	92	99
2	11	16	28	75
12	18	73	80	89
36	47	50	67	94
15	23	44	53	84
10	18	39	60	76
24	86			

11.4.2. Partición por contenido

La partición del archivo de entrada se realiza en función del contenido de uno o más campos del registro; así, por ejemplo, si se supone un archivo `f` que se desea dividir en dos archivos `f1` y `f2`, tal que `f1` tenga todos los registros que contengan en el campo clave `c`, el valor `v` y en el archivo `f2` los restantes registros.

Cristian Pilapanta 1095211-2202112332

El algoritmo de partición se muestra a continuación:

```
algoritmo particiona_contenido
....
inicio
   abrir (f, l, 'nombre')
   crear (f1, 'nombre1')
   abrir (f1, e, 'nombre1')
   crear (f2, 'nombre2')
   abrir (f2, e, 'nombre2')
   leer(v)
   mientras NO FDA (f) hacer
      leer_reg (f, r)
      si v = r.c entonces
         escribir_reg (f1,r)
      si_no
         escribir_reg (f2, r)
      fin_si
   fin_mientras
   cerrar (f, f1, f2)
fin
```

11.4.3. Selección por sustitución

La clasificación interna vista en el apartado 11.4.1 no tiene en cuenta la ventaja que supone cualquier ordenación parcial que pueda existir en el archivo de entrada. El algoritmo de selección por sustitución tiene en cuenta tal ordenación. Los pasos a dar para obtener particiones ordenadas son:

1. Leer N registros del archivo desordenado, poniéndolos todos a no congelados.
2. Obtener el registro R con clave más pequeña de entre los no congelados y escribirlo en partición.
3. Sustituir el registro por el siguiente del archivo de entrada. Este registro se congelará si su clave es más pequeña que la del registro R y no se congelará en otro caso. Si hay registro sin congelar volver al paso 2.
4. Comenzar nueva partición. Si se ha llegado a fin de fichero se repite el proceso sin leer.

Nota: Al final de este método los ficheros con las particiones tienen secuencias ordenadas, lo que no quiere decir que ambos hayan quedado completamente ordenados.

F:	3	31	14	42	10	15	8	13	63	18	50
				F1							F2
3	31	14	42	3			13	50	8	18	8
10	31	14	42	10			13	50	8	18	13
15	31	14	42	14			13	50	8	18	18
15	31	8	42	15			13	50	8	18	50
13	31	8	42	31			13	50	8	18	
13	63	8	42	42							
13	63	8	18	63							
13	50	8	18								

```
algoritmo partición_s
const n= <valor>
tipo
   registro: datos_personales
```

Cristian Pilapanta 1095211-2202112332

```
      <tipo_dato>:c
      ...
   fin_registro
   registro: datos
      datos_personales : dp
      lógico           : congela
   fin_registro
   array[1..n] de datos: arr
   archivo_s de datos_personales: arch
var
   datos_personales: r
   arr              : a
   arch             : f1,f2, f
   lógico           : sw
   entero           : númcongelados, y, posiciónmenor
inicio
      abrir(f, l, 'nombre')
      crear(f1, 'nombre1')
      abrir(f1, e, 'nombre1')
      crear(f2,'nombre2')
      abrir(f2, e, 'nombre32')
      númcongelados ← 0
      desde i ← 1 hasta n hacer
         si no fda(f) entonces
            leer_reg(f, r)
            a[i].dp ← r
            a[i].congela ← falso
         si_no
            a[i].congela ← verdad
            númcongelados ← númcongelados + 1
         fin_si
      fin_desde
      sw ← verdad
      mientras no fda(f) hacer
         mientras (númcongelados < n) y no fda(f) hacer
            buscar_no_congelado_menor(a, posiciónmenor)
            si sw entonces
               escribir_reg(f1, a[posiciónmenor].dp)
            si_no
               escribir_reg(f2, a[posiciónmenor].dp)
            fin_si
            leer_reg(f, r)
            si r.c. > a[posiciónmenor].dp.c entonces
               a[posiciónmenor].dp ← r
            si_no
               a[posiciónmenor].dp ← r
               a[posiciónmenor].congela ← verdad
               númcongelados ← númcongelados + 1
            fin_si
         fin_mientras
         sw ← no sw
         descongelar(a)
```

Cristian Pilaparda 1095211-2202112332

```
      númcongelados ← 0
   fin_mientras
   mientras númcongelados < n hacer
      buscar_no_congelado_menor(a, posiciónmenor)
      si sw entonces
         escribir_reg(f1,a[posiciónmenor].dp)
      si_no
         escribir_reg(f2, a[posiciónmenor].dp)
      fin_si
      a[posiciónmenor].congela ← verdad
      númcongelados ←   númcongelados + 1
   fin_mientras
   cerrar(f, f1, f2)
fin
```

11.4.4. Partición por secuencias

Los registros se dividen en secuencias alternativas con longitudes iguales o diferentes según los casos.

Las secuencias pueden ser de diferentes diseños:

- El archivo f se divide en dos archivos, f1 y f2, copiando alternativamente, en uno y otro, secuencias de registros de longitud *m*. (Algoritmo partición_1.)
- El archivo f se divide en dos archivos, f1 y f2, de modo que en f1 se copian los registros que ocupan las posiciones pares y en f2 los registros que ocupan las posiciones impares. (Algoritmo partición_2.)

```
algoritmo partición_1
tipo
   registro: datos personales
      <tipodato> : C
      ..............
   fin_registro
   archivo_s de datos_personales : arch
var
   datos_personales: r
   arch            : f, f1, f2
   lógico          : SW
   entero          : i, n
inicio
   abrir (f, l, 'nombre')
   crear (f1, 'nombre1')
   abrir (f1, e, 'nombre1')
   crear (f2, 'nombre2')
   abrir (f2, e, 'nombre2')
   i← 0
   leer (n)
   SW← verdad
   mientras NO FDA (f) hacer
      leer_reg (f, r)
      si SW entonces
         escribir_reg (f2, r)
      si_no
         escribir_reg (f2, r)
      fin_si
```

Cristian Pilaparita 1095211-2202112332

```
      i← i+1
      si i = n entonces
         SW ←  NO SW
         i ←  0
      fin_si
   fin_mientras
   cerrar (f, f1, f2)
fin
```

```
algoritmo partición_2
   tipo
   registro: datos_personales
         <tipo_dato> : C
         ..............
      fin_registro
      archivo_s de datos_personales : arch
var
   datos_personales : r
   arch             : f1, f2, f
   lógico           : SW
inicio
   abrir (f, l, 'nombre')
   crear (f1, 'nombre1')
   abrir (f1, e, 'nombre1')
   crear (f2, 'nombre2')
   abrir (f2, e, 'nombre2')
   SW← verdad
   mientras NO FDA (f) hacer
      leer_reg (f, r)
      si SW entonces
         escribir_reg (f1,r)
      si_no
         escribir_reg (f2,r)
      fin_si
      SW← NO SW
   fin_mientras
   cerrar (f, f1, f2)
fin
```

Cristian Pilaparita 1095211-2202112332

1095211220211233

11.5. Clasificación de archivos

Los archivos están *clasificados en orden ascendente* o *descendente* cuando todos sus registros están ordenados en sentido ascendente o descendente respecto al valor de un campo determinado, denominado *clave de ordenación*.

Si el archivo a ordenar cabe en la memoria central, se carga en un vector y se realiza una clasificación interna, transfiriendo el archivo ordenado al soporte externo o copiando el resultado en el archivo original si no se desea conservar.

En el caso de que el archivo no quepa en la memoria central, la clasificación se realizará sobre el archivo almacenado en un soporte externo. El inconveniente de este tipo de clasificación reside en el tiempo, que será mucho mayor debido especialmente a las operaciones entrada/salida de información que requiere la clasificación externa.

Los algoritmos de clasificación son muy variados, pero muchos de ellos se basan en procedimientos mixtos consistentes en aprovechar al máximo la capacidad de la memoria central. Como métodos de clasificación de archivos que no utilizan la memoria central y son aplicables a archivos secuenciales, se tienen la *mezcla directa* y la *mezcla natural*.

11.5.1. Clasificación por mezcla directa

El método más fácil de comprender es el denominado *mezcla directa*. Se analiza su aplicación a través de un breve ejemplo en el que se aplicará el método sobre un vector. Se puede pensar en los componentes del vector como las claves de los registros sucesivos del archivo.

El procedimiento consiste en una partición sucesiva del archivo y una fusión que produce secuencias ordenadas. La primera partición se hace para secuencias de longitud 1 utilizando dos archivos auxiliares y la fusión producirá secuencias ordenadas de longitud 2. A cada nueva partición y fusión se duplicará la longitud de las secuencias ordenadas. El método terminará cuando la longitud de la secuencia ordenada exceda la longitud del archivo a ordenar.

Consideremos el archivo:

```
F: 19    27    2    8    36    5    20    15    6
```

El archivo F se divide en dos nuevos archivos F1 y F2:

```
F1: 19   2   36   20   6
F2: 27   8   5    15
```

Ahora se funden los archivos F1 y F2, formando pares ordenados:

```
F: 19    27    2    8    5    36    15    20    6
```

Se vuelve a dividir de nuevo en partes iguales y en secuencias de longitud 2:

```
F1: 19   27   5    36   6
F2: 2    8    15   20
```

La fusión de los archivos producirá:

```
F: 2     8     19    27    5     15    20    36    6
```

La nueva partición será:

```
F1: 2   8    19   27   6
F2: 5   15   20   36
```

La nueva fusión será:

```
F: 2     5     8     15    19    20    27    36    6
```

Cada operación que trata por completo el conjunto de datos se denomina una *fase* y el proceso de ordenación se denomina *pasada*.

```
F1: 2    5    8    15   19   20   27   36
F2: 6
```

```
F: 2   5   6   8   15   19   20   27   36
```

Evidentemente, la clave de la clasificación es disminuir el número de pasadas e incrementar su tamaño; una secuencia ordenada es una que contiene sólo una pasada que, a su vez, contiene todos los elementos de la pasada.

EJEMPLO 11.2

Para la implementación de los siguientes algoritmos no se consideró la existencia de un registro especial que indicara el fin de archivo. La función `FDA(id_arch)` *retorna cierto cuando se accede al último registro.*

Si se considerase la existencia del registro especial que marca el fin de archivo se podría prescindir del uso de las variables lógicas `fin`, `fin1`, `fin2`.

Cristian Pilaquinga 1095211-2202112332
1095211220211233

```
algoritmo ord_mezcla_directa
...
procedimiento ordmezcladirecta
var
   datos_personales: r, r1, r2
   arch            : f, f1, f2
   // El tipo arch es archivo_s de datos_personales
   entero          : lgtud, long
   lógico          : sw, fin1, fin2
   entero          : i, j
inicio
   // calcularlongitud(f) es una función definida por el usuario que
   // devuelve el número de registros del archivo original
   long ← calcularlogitud(f)
   lgtud ← 1
   mientras lgtud < long hacer
      abrir(f, l, 'fd')
      crear(f1,'f1d')
      crear(f2,'f2d')
      abrir(f1, e,'f1d')
      abrir(f2, e,'f2d')
      i ← 0
      sw ← verdad
      mientras no FDA(f) hacer
         leer_reg(f, r)
         si sw entonces
            escribir_reg(f1, r)
         si_no
            escribir_reg(f2, r)
         fin_si
         i ← i + 1
         si i=lgtud entonces
            sw ← no sw
            i ← 0
         fin_si
      fin_mientras
      cerrar(f, f1, f2)
      abrir(f1, l, 'f1d')
      abrir(f2, l, 'f2d')
      crear(f, 'fd')
      abrir (f, e, 'fd')
      i ← 0
      j ←  0
      fin1 ← falso
      fin2 ← falso
      si FDA(f1) entonces
         fin1 ← verdad
      si_no
         leer_reg(f1, r1)
      fin_si
      si FDA(f2) entonces
         fin2 ← verdad
```

Cristian Pilapanta 1095211-2202112332

```
      si_no
         leer_reg(f2, r2)
      fin_si
      mientras no fin1 o no fin2 hacer
         mientras no fin1 y no fin2 y (i<lgtud) y (j<lgtud) hacer
            si menor(r1, r2) entonces
               escribir_reg(f, r1)
               si FDA(f1) entonces
                  fin1 ← verdad
               si_no
                  leer_reg(f1, r1)
               fin_si
               i ← i + 1
            si_no
               escribir_reg(f, r2)
               si FDA(f2) entonces
                  fin2 ← verdad
               si_no
                  leer_reg(f2, r2)
               fin_si
               j ← j + 1
            fin_si
         fin_mientras
         mientras no fin1 y (i < lgtud) hacer
            escribir_reg(f, r1)
            si FDA(f1) entonces
               fin1 ← verdad
            si_no
               leer_reg(f1, r1)
            fin_si
            i ← i + 1
         fin_mientras
         mientras no fin2 y (j < lgtud) hacer
            escribir_reg(f, r2)
            si FDA(f2) entonces
               fin2 ← verdad
            si_no
               leer_reg(f2, r2)
            fin_si
            j ← j + 1
         fin_mientras
         i ← 0
         j ← 0
      fin_mientras              // del mientras no fin1 o no fin2
      cerrar(f, f1, f2)
      lgtud ← lgtud*2
   fin_mientras                 // del mientras lgtud < long
   borrar('f1d')
   borrar('f2d')
fin_procedimiento
```

Cristian Pilaparia 1095211-2202112332

11.5.2. Clasificación por mezcla natural

Es uno de los mejores métodos de ordenación de ficheros secuenciales. Consiste en aprovechar la posible ordenación interna de las secuencias del archivo (F), obteniendo con ellas particiones ordenadas de longitud variable sobre una serie de archivos auxiliares, en este caso dos, F1 y F2. A partir de estos ficheros auxiliares se escribe un nuevo F mezclando los segmentos crecientes máximos de cada uno de ellos.

EJEMPLO 11.3

Clasificar el vector:

```
F: 19    27    2    8    36    5    20    15    6
```

Se divide F en dos vectores F1 y F2, donde se ponen alternativamente los elementos F1 y F2. F está ahora vacío.

Etapa 1, fase 1:

```
F1:   19   27/   5    20/   6
F2:    2   8     36/  15
```

Se selecciona el elemento más pequeño de F1 y F2, que pasan a estar en F3.

Etapa 1, fase 2:

```
F1:   19   27/   5   20/   6
F2:    8   36/  15
F3:    2
```

Ahora se comparan 8 y 19, se selecciona 8. De modo similar, 19 y 27:

```
F1:   5   20/   6
F2:  36/  15
F3:   2    8    19   27
```

En F1 se ha interrumpido la secuencia creciente y se continúa con F2 hasta que también en él se termine la secuencia creciente.

```
F1:   5  20/   6
F2:  15
F3:   2   8    19   27   36
```

Ahora 5 y 15 son menores que 36. Finalmente se tendrá:

```
F3: 2    8    19    27    36/   5    15    20/   6
```

F1 y F2 están ahora vacíos.

Etapa 2, fase 1:

Dividir F3 en dos:

```
F1:  2    8    19   27   36/   6
F2:  5   15    20
```

Cristian Pilaparita 1095211-2202112332
1095211220211233

Etapa 2, fase 2:

Se mezclan F1 y F2

```
F3: 2    5    8    15    19    20    27    36    6
```

Etapa 3, fase 1:

```
F1: 2    5    8    15    19    20    27    36
F2: 6
```

Etapa 3, fase 2:

```
F3: 2    5    6    8    15    19    20    27    36
```

y el archivo F3 ya está ordenado.

Algoritmo

```
algoritmo ord_mezcla_natural
   ...
procedimiento ordmezclanatural
var
      datos_personales: r, r1, r2, ant, ant1, ant2
      arch            : f, f1, f2
      //El tipo arch es archivo_s de datos_personales
      lógico          : ordenado, crece, fin, fin1, fin2
      entero          : númsec
inicio
   ordenado ← falso
   mientras no ordenado hacer
      // Partir
      abrir(f, l, 'fd')
      crear(f1, 'f1d')
      crear(f2, 'f2d');
      abrir(f1, e, 'f1d')
      abrir(f2, e, 'f2d')
      fin ← falso
      si FDA(f) entonces
         fin ← verdad
      si_no
         leer_reg(f, r)
      fin_si
      mientras no fin hacer
         ant ← r
         crece ← verdad
         mientras crece y no fin hacer
            si menorigual(ant, r) entonces
               escribir_reg(f1, r)
               ant ← r
               si FDA(f) entonces
                  fin ← verdad
```

Cristian Pilaquinta 1095211-2202112332

```
            si_no
               leer_reg(f, r)
            fin_si
         si_no
            crece ← falso
         fin_si
      fin mientras
      ant ← r
      crece ← verdad
      mientras crece y no fin hacer
      si menorigual(ant, r) entonces
            escribir_reg(f2, r)
            ant ← r
            si FDA(f) entonces
               fin ← verdad
            si_no
               leer_reg(f, r)
            fin_si
      si_no
         crece ← falso
      fin_si
   fin_mientras
fin_mientras
cerrar(f, f1, f2)
//Mezclar
abrir(f1,l, 'f1d')
abrir(f2,l, 'f2d')
crear(f, 'fd')
abrir(f, e, 'fd')
fin1 ← falso
fin2 ← falso
si FDA(f1) entonces
      fin1 ← verdad
si_no
      leer_reg(f1, r1)
fin_si
si FDA(f2) entonces
   fin2 ← verdad
si_no
   leer_reg(f2, r2)
fin_si
númsec ← 0
mientras NO fin1 y NO fin2 hacer
   ant1 ← r1
   ant2 ← r2
   crece ← verdad
   mientras NO fin1 y NO fin2 y crece hacer
      si menorigual(ant1, r1) y menorigual(ant2, r2) entonces
         si menorigual(r1, r2) entonces
            escribir_reg(f, r1)
            ant1 ← r1
            si FDA(f1) entonces
```

Cristian Pilapanta 1095211-2202112332
1095211220211233

```
            fin1 ← verdad
         si_no
            leer_reg(f1, r1)
         fin_si
      si_no
         escribir_reg(f, r2)
         ant2 ← r2
         si FDA(f2) entonces
            fin2 ← verdad
         si_no
            leer_reg(f2, r2)
         fin_si
      fin_si
   si_no
      crece ← falso
   fin_si
fin_mientras
mientras NO fin1 y menorigual(ant1, r1) hacer
   escribir_reg(f, r1)
   ant1 ← r1
   si FDA(f1) entonces
      fin1 ← verdad
   si_no
      leer_reg(f1, r1)
   fin_si
fin_mientras
mientras NO fin2 y menorigual(ant2, r2) hacer
   escribir_reg(f,r2)
   ant2 ← r2
   si FDA(f2) entonces
      fin2 ← verdad
   si_no
      leer_reg(f2, r2)
   fin_si
   fin_mientras
   númsec ← númsec + 1
fin_mientras                    // del mientras no fin1 y no fin2
si NO fin1 entonces
   númsec ← númsec+1
   mientras NO fin1 hacer
      escribir_reg(f, r1)
      si FDA(f1) entonces
         fin1 ← verdad
      si_no
         leer_reg(f1, r1)
      fin_si
   fin_mientras
fin_si
si no fin2 entonces
   númsec ← númsec+1
   mientras no fin2 hacer
      escribir_reg(f, r2)
```

Cristian Pilapanta 1095211-2202112332

```
            si FDA(f2) entonces
               fin2 ← verdad
            si_no
               leer_reg(f2, r2)
            fin_si
         fin_mientras
      fin_si
      cerrar(f, f1, f2)
      si númsec <= 1 entonces
         ordenado ← verdad
      fin_si
   fin_mientras                  // del mientras no ordenado
   borrar('f1d')
   borrar('f2d')
fin_procedimiento
```

11.5.3. Clasificación por mezcla de secuencias equilibradas

Este método utiliza la memoria de la computadora para realizar clasificaciones internas y cuatro archivos secuenciales temporales para trabajar.

Suponga un archivo de entrada F que se desea organizar por orden creciente de las claves de sus elementos. Se dispone de cuatro archivos secuenciales de trabajo, F1, F2, F3 y F4, y que se pueden colocar m elementos en la memoria central en un momento dado en una tabla T de m elementos. El proceso es el siguiente:

1. Lectura de archivo de entrada por bloques de n elementos.
2. Ordenación de cada uno de estos bloques y escritura alternativa sobre F1 y F2.
3. Fusión de F1 y F2 en bloques de $2n$ elementos que se escriben alternativamente sobre F3 y F4.
4. Fusión de F3 y F4 y escritura alternativa en F1 y F2, de bloques con $4n$ elementos ordenados.
5. El proceso consiste en doblar cada vez el tamaño de los bloques y utilizando las parejas (F1, F2) y (F3, F4).

Fichero de entrada

```
46  66  4  12   7   5  34  32  68   8  99  16  13  14  12  10
F1      [4  12  46  66]                     [ 8  16  68  99 ]
F2      [5   7  32  34]                     [10  12  13  14 ]
F3 vacío
F4 vacío
```

Fusión por bloques

```
F1 vacío
F2 vacío
F3   4   5   7   12   32   34   46   66   /F4   8   10   12   13   14   16   68   99
```

La mezcla o fusión final es

```
F1   4   5   7   8   10   12   12   13   14   16   32   34   46   66   68   99
F2 vacío
F3 vacío
F4 vacío
```

Cristian Pilapanta 1095211-2202112333

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

11.1. *Realizar el algoritmo de partición de un archivo F en dos, F1 y F2, según el contenido de un campo clave C. El contenido debe tener el valor v.*

```
algoritmo Partición_contenido
...
inicio
   abrir(f, l, 'f0')     //lectura
   crear (f1, 'f1')
   crear (f2, 'f2')
   abrir (f1, e, 'f1')   //escritura
   abrir (f2, e, 'f2')   //escritura
   mientras no fda(f) hacer
      leer_reg(f, r)
      si r.c = v entonces
         escribir_reg(f1, r)
      si_no
         escribir_reg(f2, r)
      fin_si
   fin_mientras
   cerrar(f1, f2, f)
fin
```

11.2. *Realizar el algoritmo de partición por secuencias alternativas de longitud* n.

- *Entrada: Archivo* F
- *Salida: Archivos* F1, F2
- *Secuencia: longitud* n *(registros en cada secuencia)*
- *Cada* n *registros se almacenan alternativamente en* F1 *y* F2.

```
algoritmo Partir_alternativa
tipo
   registro: reg
      ...
   fin_registro
   archivo_s de reg : arch
var
   lógico: sw
   entero: i     //contador de elementos de la secuencia
   arch: f, f1, f2
   reg: r
inicio
   abrir(f, l, 'f0')     //lectura
   crear (f1, 'f1')
   crear (f2, 'f2')
   abrir (f1, e, 'f1') //escritura
   abrir (f2, e, 'f2') //escritura
   sw ← falso
   i ← 0
   mientras no fda(f) hacer
      leer_reg(f, r)
      si NO sw entonces
         escribir_reg(f1, r)
```

Cristian Pilaquita 1095211-2202112333

```
            si_no
               escribir_reg(f2, r)
            fin_si
            i ← i+1
            si i= n entonces
               sw ← verdadero
               i ← 0        //se inicializa el contador de la secuencia
            fin_si
         fin_mientras
         cerrar(f, f1, f2)
      fin
```

11.3. *Aplicar el algoritmo de mezcla directa al archivo* F *de claves.*

F: *9 7 2 8 16 15 2 10*

1. Primera división (partición en secuencias de longitud 1):

   F1: 9 2 16 2

   F2: 7 8 15 10

2. Mezcla de F1 y F2, formando pares ordenados:

   F: 7 9 || 2 8 || 15 16 || 2 10

3. Segunda división (partición en secuencias de longitud 2):

   F1: 7 9 || 15 16

   F2: 2 8 || 2 10

4. Mezcla de F1 y F2:

   F: 2 7 8 9 || 2 10 15 16

5. Tercera división (partición en secuencias de longitud 4):

   F1: 2 7 8 9

   F2: 2 10 15 16

6. Mezcla de F1 y F2 (última):

   F: 2 2 7 8 9 10 15 16

11.4. *Escribir el procedimiento de mezcla de dos archivos ordenados en secuencias de una determinada longitud.*

```
procedimiento fusión (E cadena: nombre1, nombre2, nombre3; E entero: lgtud)
   var
      arch : f1, f2, f
      //el tipo arch se supone definido en el programa principal
      reg: r1, r2
      // el tipo reg se supone definido en el programa principal
      entero : i, j
      lógico : fin1, fin2
inicio
      {los nombres de los archivos en el dispositivo de almacenamiento se
      pasan al procedimiento de fusión a través de las variables nombre1,
      nombre2 y nombre3 }
   abrir(f1, l, nombre1)
   abrir(f2, l, nombre2)
   crear(f, nombre3)
   abrir (f, e, nombre3)
   leerRegYFin(f1, r1, fin1)
   {leerRegYFin es un procedimiento, desarrollado más adelante, que
      lee un registro y detecta la marca de fin de archivo}
```

Cristian Pilapaña 1095211-2202112333
1095211220211233

```
   leerRegYFin(f2, r2, fin2)
   mientras no fin1 o no fin2 hacer
      i ← 0
      j ← 0
      mientras no fin1 y no fin2 y (i<lgtud) y (j<lgtud) hacer
         //lgtud es la longitud de la secuencia recibida como parámetro
         si menor(r1,r2) entonces
            escribir_reg(f, r1)
            leerRegYFin(f1, r1, fin1)
            i ← i+1
         si_no
            escribir_reg(f, r2)
            leerRegYFin(f2, r2, fin2)
            j ← j+1
         fin_si
      fin_mientras
      mientras no fin1 y (i < lgtud) hacer
         escribir_reg(f, r1)
         leerRegYFin(f1, r1, fin1)
         i       ← i+1
      fin_mientras
      mientras no fin2 y (j < lgtud) hacer
         escribir_reg(f, r2)
         leerRegYFin(f2, r2, fin2)
         j       ← j+1
      fin_mientras
   fin_mientras            // del mientras no fin1 o no fin2
   cerrar(f, f1, f2)
fin_procedimiento
procedimiento leerRegYFin(E/S arch: f; E/S reg: r; E/S lógico: fin)
inicio
   si fda(f) entonces
      fin ← verdad
   si_no
      leer_reg(f, r)
   fin_si
fin_procedimiento
```

11.5. *Escribir el procedimiento de ordenación por mezcla directa de un archivo con* `long` *registros, utilizando el procedimiento fusión del ejercicio anterior.*

```
procedimiento ordenarDirecta(E cadena: nombref, nombref1, nombref2;
                     E entero: long)
var
   entero: lgtud
inicio
   lgtud ← 1
   mientras lgtud <= long hacer
     partirAlternativoEnSec (nombref, nombref1, nombref2, lgtud)
     fusión(nombref1, nombref2, nombref, lgtud)
     lgtud ← lgtud * 2
   fin_mientras
   borrar(nombref1)
   borrar(nombref2)
```

Cristian Pilapanta 1095211-2202112333

```
fin_procedimiento
procedimiento partirAlternativoEnSec (E cadena: nombref, nombref1,
                                      nombref2; E entero: lgtud)
var
   lógico: sw
   entero: i            // contador de elementos de la secuencia
   arch: f1, f2, f      // tipo definido en el programa principal
   reg: r               // tipo definido en el programa principal
inicio
   abrir(f, l, nombref)   //lectura
   crear (f1, nombref1)
   crear (f2, nombref2)
   abrir (f1, e, nombref1) //escritura
   abrir (f2, e, nombref2) //escritura
   sw ← falso
   i ← 0
   mientras no fda(f) hacer
      leer_reg(f, r)
      si NO sw entonces
         escribir_reg(f1, r)
      si_no
         escribir_reg(f2, r)
      fin_si
      i ← i+1
      si i= lgtud entonces
         sw ← verdadero
         i ← 0        //se inicializa el contador de la secuencia
      fin_si
   fin_mientras
   cerrar(f, f1, f2)
fin
```

11.6. *Escribir el procedimiento de ordenación por mezcla natural de un archivo, utilizando los procedimientos auxiliares* `partir` *y* `mezclar` *que se suponen implementados. El procedimiento* `partir` *aprovecha las secuencias ordenadas que pudieran existir en el archivo original y las coloca alternativamente sobre dos archivos auxiliares. El procedimiento* `mezclar` *construye a partir de dos ficheros auxiliares uno nuevo, mezclando las secuencias crecientes que encuentra en los ficheros auxiliares para construir sobre el destino secuencias crecientes de longitud mayor.*

```
procedimiento ordenarNatural(E cadena: nombref, nombref1, nombref2)
var
   entero: númsec
   lógico: ordenado
inicio
   ordenado ← falso
   mientras NO ordenado hacer
      partir (nombref, nombref1, nombref2)
      númsec ← 0
      mezclar(nombref1, nombref2, nombref, númsec)
      si númsec <=1 entonces
         ordenado ← verdad
      fin_si
```

Cristian Pilapanta 1095211-2202112333
1095211220211233

```
    fin_mientras
    borrar(nombref1)
    borrar(nombref2)
fin_procedimiento
```

CONCEPTOS CLAVE

- Mezcla
- Mezcla directa
- Mezcla natural
- Ordenación externa
- Partición

RESUMEN

La ordenación externa se emplea cuando la masa de datos a procesar es grande y no cabe en la memoria central de la computadora. Si el archivo es directo, aunque los registros se encuentran colocados en él de forma secuencial, servirá cualquiera de los métodos de clasificación vistos como métodos de ordenación interna, con ligeras modificaciones debido a las operaciones de lectura y escritura de registros en el disco. Si el archivo es secuencial, es necesario emplear otros métodos basados en procesos de partición y medida.

1. La partición es el proceso por el cual los registros de un archivo se reparten en otros dos o más archivos en función de una condición.
2. La fusión o mezcla consiste en reunir en un archivo los registros de dos o más. Habitualmente los registros de los archivos originales se encuentran ordenados por un campo clave, y la mezcla ha de efectuarse de tal forma que se obtenga un archivo ordenado por dicho campo clave.
3. Los archivos están *clasificados en orden ascendente o descendente* cuando todos sus registros están ordenados en sentido ascendente o descendente respecto al valor de un campo determinado, denominado *clave de ordenación*. Los algoritmos de clasificación son muy variados: 1) si el archivo a ordenar cabe en la memoria central, se carga en un vector y se realiza una clasificación interna, transfiriendo a continuación el archivo ordenado al soporte externo; 2) si el archivo a ordenar no cabe en memoria central y es secuencial son aplicables la mezcla directa y la mezcla natural; 3) si no es secuencial pueden aplicarse métodos similares a los vistos en la clasificación interna con ligeras modificaciones; 4) otros métodos se basan en procedimientos mixtos consistentes en aprovechar al máximo la capacidad de la memoria central.
4. La clasificación por *mezcla directa* consiste en una partición sucesiva del archivo y una fusión que produce secuencias ordenadas. La primera partición se hace para secuencias de longitud 1 y la fusión producirá secuencias ordenadas de longitud 2. A cada nueva partición y fusión se duplicará la longitud de las secuencias ordenadas. El método terminará cuando la longitud de la secuencia ordenada exceda la longitud del archivo a ordenar.
5. La clasificación por mezcla natural consiste en aprovechar la posible ordenación interna de las secuencias del archivo original (F), obteniendo con ellas particiones ordenadas de longitud variable sobre los ficheros auxiliares. A partir de estos ficheros auxiliares escribiremos un nuevo F mezclando los segmentos crecientes de cada uno de ellos.
6. La búsqueda es el proceso de localizar un registro en un archivo con un determinado valor en uno de sus campos. Los archivos de tipo secuencial obligan a efectuar búsquedas secuenciales, mientras que los archivos directos son estructuras de acceso aleatorio y permiten otros tipos de búsquedas.
7. La búsqueda binaria podría aplicarse a archivos directos con los registros colocados uno a continuación de otro y ordenados por el campo por el que se desea efectuar la búsqueda.

Cristian Pilaparita 1095211-2202112333

EJERCICIOS

11.1. Se desea intercalar los registros del archivo P con los registros del Q y grabarlos en otro archivo R.

Nota: Los archivos P y Q están clasificados en orden ascendente por una determinada clave y se desea que el archivo R quede también ordenado en modo ascendente.

11.2. Los archivos M, N y P contienen todas las operaciones de ventas de una empresa en los años 1985, 1986 y 1987, respectivamente. Se desea un algoritmo que intercale los registros de los tres archivos en un solo archivo Z, teniendo en cuenta que los tres están clasificados en orden ascendente por el campo clave ventas.

11.3. Se dispone de dos archivos secuenciales F1 y F2 que contienen los mismos campos. Los dos archivos están ordenados de modo ascendente por el campo clave (alfanumérico) y existen registros comunes a ambos. Se desea diseñar un programa que obtenga: *a*) un archivo C a partir de F1 y F2, que contenga todos los registros comunes, pero sólo una vez; *b*) un archivo que contenga todos los registros que no son comunes a F1 y F2.

11.4. Se desea intercalar los registros del archivo A con los registros del B y grabarlos en un tercer archivo C. Los archivos A y B están clasificados en orden ascendente por su campo clave. Y se desea también que C quede clasificado en orden ascendente.

11.5. El archivo A contiene los números de socios del Club Deportivo Esmeralda y B los códigos de los socios del Club Deportivo Diamante. Se desea crear un archivo C que contenga los números de los socios que pertenecen a ambos clubes. Asimismo, se desea saber cuántos registros se han leído y cuántos se han grabado.

11.6. Los archivos F1, F2 y F3 contienen todas las operaciones de ventas de una compañía informática en los años 1985, 1986 y 1987, respectivamente. Se desea un programa que intercale todos los registros de los tres en un solo archivo F, suponiendo que todo registro posee un campo clave y que F1, F2 y F3 están clasificados en orden ascendente de ese campo clave.

11.7. Se desea actualizar un archivo maestro de la nómina de la compañía Aguas del Pacífico con un archivo MODIFICACIONES que contiene todas las incidencias de empleados (altas, bajas, modificaciones). Ambos archivos están clasificados en orden ascendente del código de empleado (campo clave). El nuevo archivo maestro actualizado debe conservar la clasificación ascendente por código de empleado y sólo debe existir un registro por empleado.

11.8. Se tiene un archivo maestro de inventarios con los siguientes campos:

```
CÓDIGO DE ARTÍCULO      DESCRIPCIÓN
EXISTENCIAS
```

Se desea actualizar el archivo maestro con los movimientos habidos durante el mes (altas/bajas). Para ello se incluyen los movimientos en un archivo OPERACIONES que contiene los siguientes campos:

```
CÓDIGO DE ARTÍCULO      CANTIDAD
OPERACIÓN (1-Alta, 2-Baja)
```

Los dos archivos están clasificados por el mismo campo clave.

Cristian Pilaparita 1095211-2202112333

CAPÍTULO 12

Estructuras dinámicas lineales de datos (pilas, colas y listas enlazadas)

12.1. Introducción a las estructuras de datos
12.2. Listas
12.3. Listas enlazadas
12.4. Procesamiento de listas enlazadas
12.5. Listas circulares
12.6. Listas doblemente enlazadas
12.7. Pilas
12.8. Colas
12.9. Doble cola
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Cristian Pilaparia 1095211-2202112333

1095211220211233

INTRODUCCIÓN

Los datos estudiados hasta ahora se denominan *estáticos*. Ello es debido a que las variables son direcciones simbólicas de posiciones de memoria; esta relación entre nombres de variables y posiciones de memoria es una *relación estática* que se establece por la declaración de las variables de una unidad de programa y permanece durante la ejecución de esa unidad. Aunque el contenido de una posición de memoria asociada con una variable puede cambiar durante la ejecución, es decir, el valor de la variable puede cambiar, las variables por sí mismas no se pueden crear ni destruir durante la ejecución. En consecuencia, se denominan *variables estáticas*.

En algunas ocasiones, sin embargo, no se conoce por adelantado cuánta memoria se requerirá para un programa. En esos casos es conveniente disponer de un método para adquirir posiciones adicionales de memoria a medida que se necesiten durante la ejecución del programa y liberarlas cuando no se necesitan. Las variables que se crean y están disponibles durante la ejecución de un programa se denominan *variables dinámicas*; y se representan con un tipo de datos conocido como *puntero*. Se utilizan para crear *estructuras dinámicas de datos* que se pueden ampliar y comprimir a medida que se requieran durante la ejecución del programa. Una estructura de datos dinámica es una colección de elementos denominados nodos de la estructura —normalmente de tipo registro— que son enlazados juntos. Las estructuras dinámicas de datos se clasifican en lineales y no lineales. El estudio de las estructuras lineales, *listas*, *pilas* y *colas*, es el objetivo de este capítulo.

12.1. Introducción a las estructuras de datos

En capítulos anteriores se ha introducido a las estructuras de datos, definiendo tipos y estructuras de datos primitivos, tales como *enteros, real* y *carácter*, utilizados para construir tipos más complicados como *arrays* y *registros*, denominados estructuras de datos compuestos. Tienen una estructura, porque sus datos están relacionados entre sí. Las estructuras compuestas, como *arrays* y *registros*, están soportadas en la mayoría de los lenguajes de programación, debido a que son necesarias en casi todas las aplicaciones.

La potencia y flexibilidad de un lenguaje está directamente relacionada con las estructuras de datos que posee. La programación de algoritmos complicados puede resultar muy difícil en un lenguaje con estructuras de datos limitados, caso de Fortran y Cobol. En ese caso es conveniente pensar en la implementación con lenguajes que soporten punteros como **C** y **C++** o bien que no soporten pero tengan recolección de basura como **Java** o **C#**, o bien recurrir, al menos en el periodo de formación, al clásico Pascal sin olvidar Python.

Cuando una aplicación particular requiere una estructura de datos no soportada por el lenguaje, se hace necesaria una labor de programación para representarla. Se dice que necesitamos *implementar* la estructura de datos. Esto evidentemente significa más trabajo para el programador. Si la programación no se hace bien, se puede malgastar tiempo de programación y —naturalmente— de computadora. Por ejemplo, supongamos que tenemos un lenguaje como Pascal que permite arrays de una dimensión de números enteros y reales, pero no arrays multidimensionales. Para implementar una tabla con cinco filas y diez columnas podemos utilizar:

```
type
   array[0..10] of real: FILA;
var
   FILA: FILA1, FILA2, FILA3, FILA4, FILA5;
```

La llamada al elemento de la tercera fila y sexta columna se realizará con la instrucción:

```
FILA3 [6]
```

Un método muy eficaz es diseñar procedimientos y funciones que ejecuten las operaciones realizadas por las estructuras de datos. Sin embargo, con las estructuras vistas hasta ahora arrays y registros tienen dos inconvenientes: 1) *la reorganización de una lista, si ésta implica movimiento de muchos elementos de datos, puede ser muy costosa*, y 2) *son estructuras de datos estáticas*.

Una estructura de datos es *estática* cuando el tamaño ocupado en memoria es fijo, es decir, siempre ocupa la misma cantidad de espacio en memoria. Por consiguiente, si se representa una lista como vector, se debe anticipar (*declarar* o *dimensionar*) la longitud de esa lista cuando se escribe un programa; es imposible ampliar el espacio de memoria disponible (algunos lenguajes permiten dimensionar dinámicamente el tamaño de un array durante la ejecución del programa, como es el caso de Visual Basic). En consecuencia, puede resultar difícil representar diferentes estructuras de datos.

Los arrays unidimensionales son estructuras estáticas lineales ordenadas secuencialmente. Las estructuras se convierten en dinámicas cuando los elementos pueden ser insertados o suprimidos directamente sin necesidad de algoritmos complejos. Se distinguen las estructuras dinámicas de las estáticas por los modos en que se realizan las inserciones y borrados de elementos.

12.1.1. Estructuras dinámicas de datos

Las estructuras dinámicas de datos "crecen a medida que se ejecuta un programa". Una *estructura dinámica de datos* es una colección de elementos —llamados *nodos*— que son normalmente registros. Al contrario que un array, que contiene espacio para almacenar un número fijo de elementos, una estructura dinámica de datos se amplía y contrae durante la ejecución del programa, basada en los registros de almacenamiento de datos del programa.

Las estructuras dinámicas de datos se pueden dividir en dos grandes grupos:

lineales
- pilas
- colas
- listas enlazadas

Cristian Pilapaña 1095211-2202112333

no lineales { árboles, grafos

Las estructuras dinámicas de datos se utilizan para almacenamiento de datos del mundo real que están cambiando constantemente. Un ejemplo típico ya lo hemos visto como estructura estática de datos: la lista de pasajeros de una línea aérea. Si esta lista se mantuviera en orden alfabético en un array, sería necesario hacer espacio para insertar un nuevo pasajero por orden alfabético. Esto requiere utilizar un bucle para copiar los datos del registro de cada pasajero en el siguiente elemento del array. Si en su lugar se utilizara una estructura dinámica de datos, los nuevos datos del pasajero se pueden insertar simplemente entre dos registros existentes sin un mínimo esfuerzo.

Las estructuras dinámicas de datos son extremadamente flexibles. Como se ha descrito anteriormente, es relativamente fácil añadir nueva información creando un nuevo nodo e insertándolo entre nodos existentes. Se verá que es también relativamente fácil modificar estructuras dinámicas de datos, eliminando o borrando un nodo existente.

En este capítulo examinaremos las tres estructuras dinámicas lineales de datos: **listas**, **colas** y **pilas**, dejando para el próximo capítulo las estructuras no lineales de datos: **árboles** y **grafos**.

Una *estructura estática de datos* es aquella cuya estructura se especifica en el momento en que se escribe el programa y no puede ser modificada por el mismo. Los valores de sus diferentes elementos pueden variar, pero no su estructura, ya que ésta es fija.

Una *estructura dinámica de datos* puede modificar su estructura mediante el programa. Puede ampliar o limitar su tamaño mientras se ejecuta el programa.

Cristian Pilaquinta 1095211-2202112333

12.2. Listas

Una *lista lineal* es un conjunto de elementos de un tipo dado que pueden variar en número y donde cada elemento tiene un único predecesor y un único sucesor o siguiente, excepto el primero y último de la lista. Ésta es una definición muy general que incluye ficheros y vectores.

Los elementos de una lista lineal se almacenan normalmente contiguos —un elemento detrás de otro— en posiciones consecutivas de la memoria. Las sucesivas entradas en una guía o directorio telefónico, por ejemplo, están en líneas sucesivas, excepto en las partes superior e inferior de cada columna. Una lista lineal se almacena en la memoria principal de una computadora en posiciones sucesivas de memoria; cuando se almacenan en cinta magnética, los elementos sucesivos se presentan en sucesión en la cinta. Esta asignación de memoria se denomina *almacenamiento secuencial*. Posteriormente se verá que existe otro tipo de almacenamiento denominado *encadenado* o *enlazado*.

Las líneas así definidas se denominan *contiguas*; las operaciones que se pueden realizar con listas lineales contiguas son:

1. Insertar, eliminar o localizar un elemento.
2. Determinar el tamaño —número de elementos— de la lista.
3. Recorrer la lista para localizar un determinado elemento.
4. Clasificar los elementos de la lista en orden ascendente o descendente.
5. Unir dos o más listas en una sola.
6. Dividir una lista en varias sublistas.
7. Copiar la lista.
8. Borrar la lista.

Una lista lineal contigua se almacena en la memoria de la computadora en posiciones sucesivas o adyacentes y se procesa como un array unidimensional. En este caso, el acceso a cualquier elemento de la lista y la adición de nuevos elementos es fácil; sin embargo, la inserción o borrado requiere un desplazamiento de lugar de los elementos que le siguen y, en consecuencia, el diseño de un algoritmo específico.

Para permitir operaciones con listas como arrays se deben dimensionar éstos con tamaño suficiente para que contengan todos los posibles elementos de la lista.

EJEMPLO 12.1

Se desea leer el elemento j-ésimo de una lista P.

El algoritmo requiere conocer el número de elementos de la lista (su longitud, L). Los pasos a dar son:

1. **conocer** longitud de la lista L.
2. **si** L = 0 visualizar «error lista vacía».
   si_no comprobar si el elemento *j*-ésimo está dentro del rango permitido de elementos 1 <= j <= L; en este caso, asignar el valor del elemento P(j) a una variable B; si el elemento *j*-ésimo no está dentro del rango, visualizar un mensaje de error «elemento solicitado no existe en la lista».
3. **fin**.

El pseudocódigo correspondiente sería:

```
procedimiento acceso(E lista: P; S elementolista: B; E entero: L, J)
inicio
   si L = 0 entonces
      escribir('Lista vacía')
   si_no
      si (j >= 1) y (j <= L) entonces
         B ← P[j]
      si_no
         escribir('ERROR: elemento no existente')
      fin_si
   fin_si
fin
```

EJEMPLO 12.2

Borrar un elemento J *de la lista* P.

Variables

- L longitud de la lista
- J posición del elemento a borrar
- I subíndice del array P
- P lista

Las operaciones necesarias son:

1. Comprobar si la lista es vacía.
2. Comprobar si el valor de J está en el rango I de la lista 1 <= J <= L.
3. En caso de J correcto, mover los elementos J+1, J+2, . . . , a las posiciones J, J+1, . . . , respectivamente, con lo que se habrá borrado el antiguo elemento J.
4. Decrementar en uno el valor de la variable L, ya que la lista contendrá ahora L - 1 elementos.

El algoritmo correspondiente será:

```
inicio
   si L = 0 entonces
```

Cristian Pilaparia 1095211-2202112333

```
      escribir('lista vacía')
   si_no
      leer(J)
      si (J >= 1) y (J <= L) entonces
         desde I ← J hasta L-1 hacer
            P[I] ← P[I+1]
         fin_desde
         L ← L-1
      si_no
         escribir('Elemento no existe')
      fin_si
   fin_si
fin
```

Una *lista contigua* es aquella cuyos elementos son adyacentes en la memoria o soporte direccionable. Tiene unos límites izquierdo y derecho o inferior/superior que no pueden ser rebajados cuando se le añade un elemento.

La inserción o eliminación de un elemento, excepto en la cabecera o final de la lista, necesita una traslación de una parte de los elementos de la misma: la que precede o sigue a la posición del elemento modificado.

Las operaciones directas de añadir y eliminar se efectúan únicamente en los extremos de la lista. Esta limitación es una de las razones por las que esta estructura es poco utilizada.

Cristian Pilaguela 1095211-2202112333

Las *listas enlazadas* o de almacenamiento enlazado o encadenado son mucho más flexibles y potentes, y su uso es mucho más amplio que las listas contiguas.

12.3. Listas enlazadas[1]

Los inconvenientes de las listas contiguas se eliminan con las listas enlazadas. Se pueden almacenar los elementos de una lista lineal en posiciones de memoria que no sean contiguas o adyacentes.

Una *lista enlazada* o *encadenada* es un conjunto de elementos en los que cada elemento contiene la posición —o dirección— del siguiente elemento de la lista. Cada elemento de la lista enlazada debe tener al menos dos campos: uno que tiene el valor del elemento y el otro campo (enlace, *link*) contiene la posición del siguiente elemento, es decir, su conexión, enlace o encadenamiento. Los elementos de una lista son enlazados por medio de los campos enlaces.

Las listas enlazadas tienen una terminología propia que suele utilizarse. Primero, los valores se almacenan en un *nodo* (figura 12.1).

Figura 12.1. Nodo con dos campos.

Una lista enlazada se muestra en la figura 12.2.

[1] Las listas enlazadas se conocen también en Latinoamérica con el término "ligadas" y "encadenadas". El término en inglés es *linked list*.

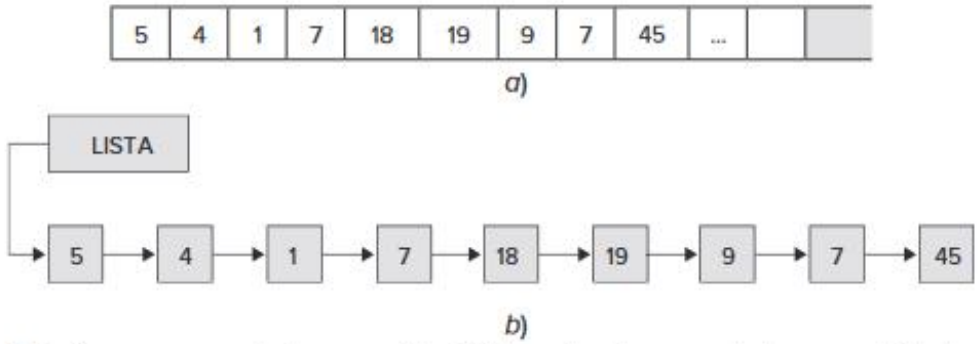


Figura 12.2. *a*) array representado por una lista; *b*) lista enlazada representada por una lista de enteros.

Los componentes de un nodo se llaman *campos*. Un nodo tiene al menos un campo *dato* o *valor* y un *enlace* (indicador o puntero) con el siguiente nodo. El campo enlace apunta (proporciona la dirección o referencia) al siguiente nodo de la lista. El último nodo de la lista enlazada, por convenio, suele representarse por un enlace con la palabra reservada `nil` (*nulo*), una barra inclinada (/) y, en ocasiones, el símbolo eléctrico de tierra o masa (figura 12.3).

Figura 12.3. Representación del último nodo de una lista.

La implementación de una lista enlazada depende del lenguaje; **C**, **C++**, **Pascal**, **PL/I**, **Ada** y **Modula-2** utilizan simplemente como enlace una *variable puntero*, o ***puntero*** (**apuntador**). Java no dispone de punteros, por consiguiente, resuelve el problema de forma diferente y almacena en el enlace la referencia al siguiente objeto nodo. Los lenguajes como Fortran y Cobol no disponen de este tipo de datos y se debe simular con una variable entera que actúa como indicador o cursor. En nuestro libro utilizaremos a partir de ahora el término *puntero* (**apuntador**) para describir el enlace entre dos elementos o nodos de una lista enlazada.

Un *puntero* (**apuntador**) es una variable cuyo valor es la dirección o posición de otra variable.

En las listas enlazadas no es necesario que los elementos de la lista sean almacenados en posiciones físicas adyacentes, ya que el puntero indica dónde se encuentra el siguiente elemento, tal como se indica en la figura 12.4.

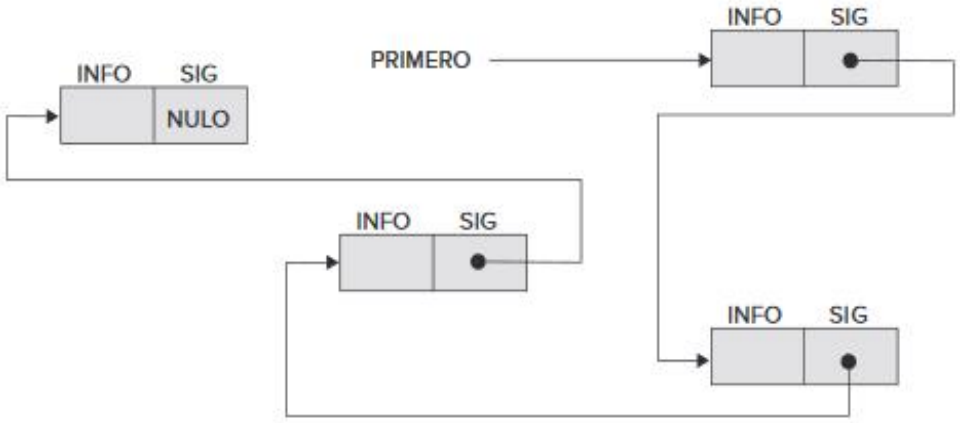


Figura 12.4. Elementos no adyacentes de una lista enlazada.

Por consiguiente, la inserción y borrado no exigen desplazamiento como en el caso de las listas contiguas; para eliminar el 45° elemento ('INÉS') de una lista lineal con 2500 elementos [figura 12.5 *a*)] sólo es necesario cambiar

Cristian Pilaquinga 1095211-2202112333

el puntero en el elemento anterior, 44°, y que apunte ahora al 46° elemento [figura 12.5 *b)*]. Para insertar un nuevo elemento (`'HIGINIO'`) después del 43° (`'GONZALO'`), es necesario cambiar el puntero del 43° y hacer que el nuevo elemento apunte al 44° [figura 12.5 *c)*].

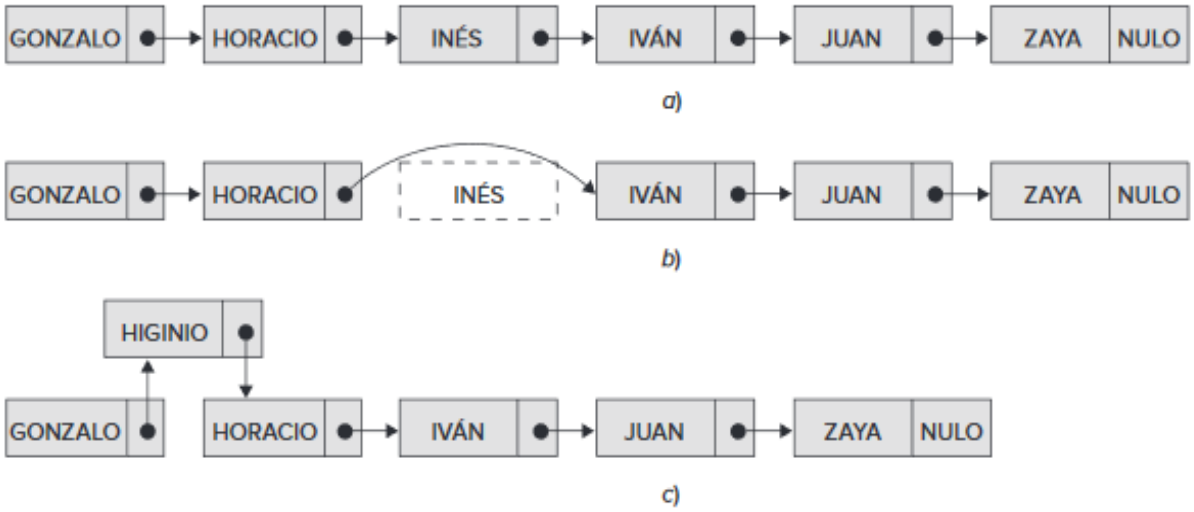


Figura 12.5. Inserción y borrado de elementos.

Una lista enlazada sin ningún elemento se llama *lista vacía*. Su puntero inicial o de cabecera tiene el valor nulo (`nil`).

Una lista enlazada se define por:

- El tipo de sus elementos: campo de información (datos) y campo enlace (*puntero o apuntador*).
- Un *puntero de cabecera* que permite acceder al primer elemento de la lista.
- Un medio para detectar el último elemento de la lista: *puntero nulo* (`nil`).

EJEMPLO 12.3

El director de un hotel desea registrar el nombre de cada cliente y el número de habitación que ocupará a medida que van llegando al hotel —el antiguo libro de entradas—. También desea disponer de una lista de sus clientes por orden alfabético en cualquier momento.

Ya que no es posible registrar los clientes alfabética y cronológicamente en la misma lista, se necesita o bien listas alfabéticas independientes o bien añadir punteros a la lista existente, con lo que sólo se utilizará una única lista. El método manual en el libro requería muchos cruces y reescrituras; sin embargo, una computadora mediante un algoritmo adecuado lo realizará fácilmente.

Por cada nodo de la lista el campo de información o datos tiene dos partes: nombre del cliente y número de habitación. Si *x* es un puntero a uno de estos nodos, *l[x].nombre* y *l[x].habitación* representarán las dos partes del campo información.

El listado alfabético se consigue siguiendo el orden de los punteros de la lista (campo puntero), se utiliza una variable `CABECERA(S)` para apuntar al primer cliente.

```
CABECERA ← 3
```

Así, `CABECERA(S)` es 3, ya que el primer cliente, `Antolín`, ocupa el lugar 3; a su vez, el puntero asociado al nodo ocupado por `Antolín` contiene el valor 10, que es el segundo nombre de los clientes en orden alfabético y éste tiene como campo puntero el valor 7, y así sucesivamente. El campo puntero del último cliente, `Tomás`, contiene el puntero nulo indicado por un `0` o bien una `Z`.

Cristian Pilaparita 1095211-2202112333

	Registro	Nombre	Habitación	Puntero
S(=3)	1	Tomás	324	z (final)
	2	Cazorla	28	8
	3	Antolín	95	10
	4	Pérez	462	6
	5	López	260	12
	6	Sánchez	220	1
	7	Bautista	115	2
	8	García	105	9
	9	Jiménez	173	5
	10	Apolinar	341	7
	11	Martín	205	4
	12	Luzárraga	420	11
	.	.	.	.
	.	.	.	.
	.	.	.	.

Figura 12.6. Lista enlazada de clientes de un hotel.

Cristian Pilapaña 1095211-2202112333

12.4. Procesamiento de listas enlazadas

Para procesar una lista enlazada se necesitan las siguientes informaciones:

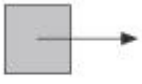
- Primer nodo (cabecera de la lista).
- El tipo de sus elementos.

Las operaciones que normalmente se ejecutan con listas incluyen:

1. Recuperar información de un nodo específico (acceso a un elemento).
2. Encontrar el nodo que contiene una información específica (localizar la posición de un elemento dado).
3. Insertar un nuevo nodo en un lugar específico de la lista.
4. Insertar un nuevo nodo en relación con una información particular.
5. Borrar (eliminar) un nodo existente que contiene información específica.

12.4.1. Implementación de listas enlazadas con punteros

Como ya hemos visto, la representación gráfica de un puntero consiste en una flecha que sale del puntero y llega a la variable dinámica apuntada.



Para declarar una variable de tipo puntero:

```
tipo
   puntero_a <tipo_dato>: punt
```

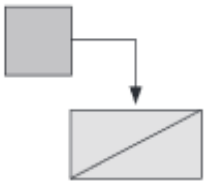
```
var
   punt : p, q
```

El <tipo_dato> podrá ser simple o estructurado.

Operaciones con punteros:

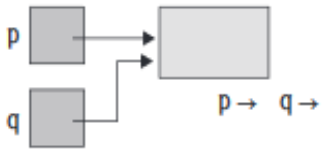
Inicialización:

p ← **nulo** A nulo para indicar que no apunta a ninguna variable.



Comparación:

p = q Con los operadores = o <>.



Asignación

p ← q Implica hacer que el puntero p apunte a donde apunta q.

Creación de variables dinámicas

Reservar(p) Reservar espacio en memoria para la variable dinámica.

Eliminación de variables dinámicas

Liberar(p) Liberar el espacio en memoria ocupado por la variable dinámica.

Variables dinámicas

Variable simple o estructura de datos sin nombre y creada en tiempo de ejecución.

Para acceder a una variable dinámica apuntada, como no tiene nombre se escribe p →

Las variables p → podrán intervenir en toda operación o expresión de las permitidas para una variable estática de su mismo tipo.

Nodo

Las estructuras dinámicas de datos están formadas por nodos.

Un nodo es una variable dinámica constituida por al menos dos campos:

- El campo dato o valor (elemento).
- El campo enlace, en este caso de tipo puntero (sig).

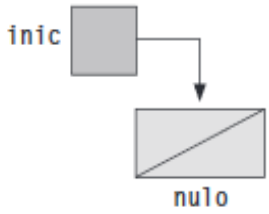
Cristian Pilapaña 1095211-2202112333

```
tipo
   registro: nodo
      //elemento es el campo que contiene la información
      <tipo_elemento>: elemento
      //punt apunta al siguiente elemento de la estructura
      punt          : sig
      (según la estructura de que se trate podrá haber uno o varios
      campos de tipo punt)
      ...           : ...
fin_registro
```

Creación de la lista

La creación de la lista conlleva la inicialización a nulo del puntero (`inic`), que apunta al primer elemento de la lista.

```
tipo
   puntero_a nodo: punt
   registro: tipo_elemento
      ... : ...
   fin_registro
   registro: nodo
      tipo_elemento : elemento
      punt          : sig
   fin_registro
var punt          : inic, posic, anterior
   tipo_elemento  : elemento
   lógico         : encontrado
inicio
   inicializar(inic)
   ...
fin

procedimiento inicializar(S punt: inic)
   inicio
      inic ← nulo
   fin_procedimiento
```



Inserción de un elemento

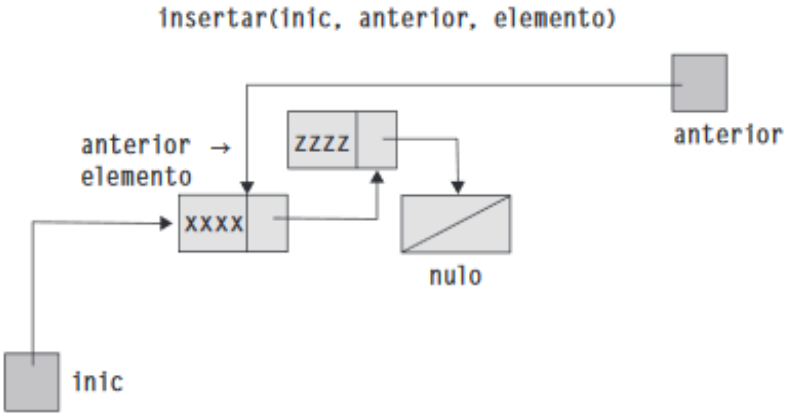
La inserción tiene dos casos particulares:

- Insertar el nuevo nodo en el frente, principio de la lista.
- Insertar el nuevo nodo en cualquier otro lugar de la lista.

El procedimiento `insertar` añade un nuevo elemento a continuación de `anterior`, si `anterior` fuera nulo significa que ha de insertarse al comienzo de la lista.

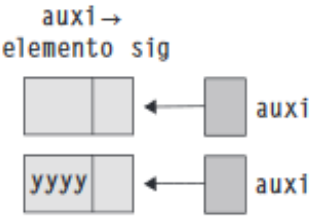
Cristian Pilaparita 1095211-2202112333
1095211220211233

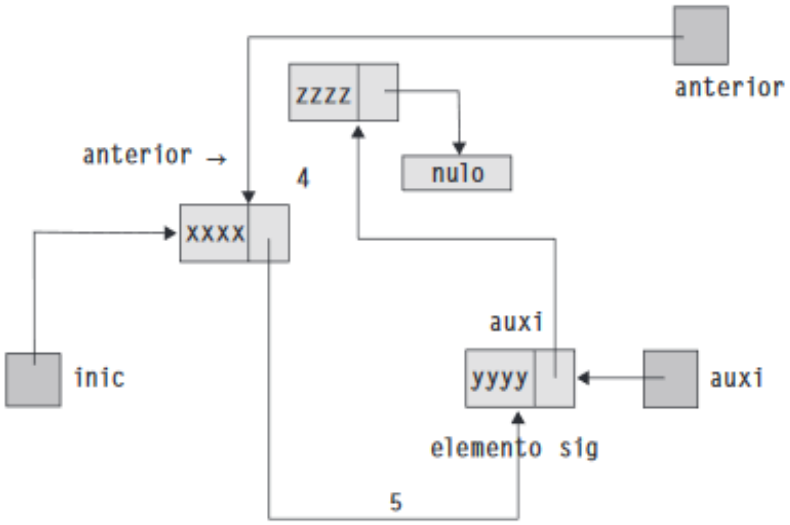
1.



2.

3.





1. Situación de partida.
2. `reservar(auxi)`.
3. Introducir la nueva información en `auxi→.elemento`.
4. Hacer que `auxi→ sig` apunte a donde lo hacía `anterior→.sig`.
5. Conseguir que `anterior→.sig` apunte a donde lo hace `auxi`.

```
procedimiento insertar(E/S punt: inic, anterior;
                       E tipo_elemento: elemento)
   var punt: auxi
   inicio
      reservar(auxi)
      auxi→.elemento ← elemento
      si anterior = nulo entonces
         auxi →.sig ← inic
```

Cristian Pilaparda 1095211-2202112333

```
        inic → auxi
      si_no
        auxi→.sig ← anterior→.sig
        anterior→.sig ← auxi
      fin_si
      anterior ← auxi    // Opcional
  fin_procedimiento
```

Eliminación de un elemento de una lista enlazada

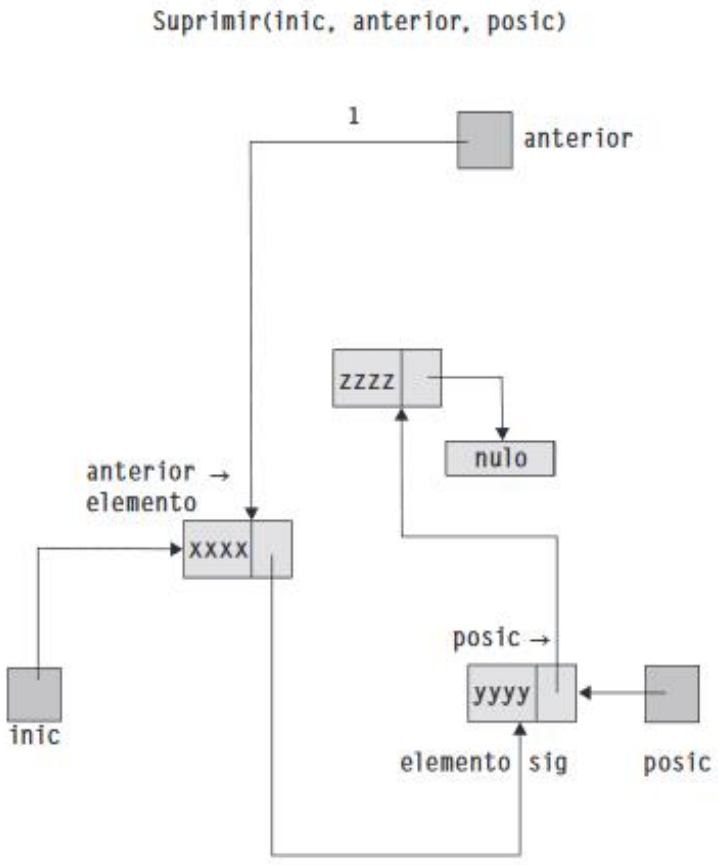
Antes de proceder a la eliminación de un elemento de la lista, deberemos comprobar que no está vacía. Para lo que podremos recurrir a la función `vacía`.

```
lógico función vacía(E punt: inic)
inicio
   devolver(inic = nulo)
fin_función
```

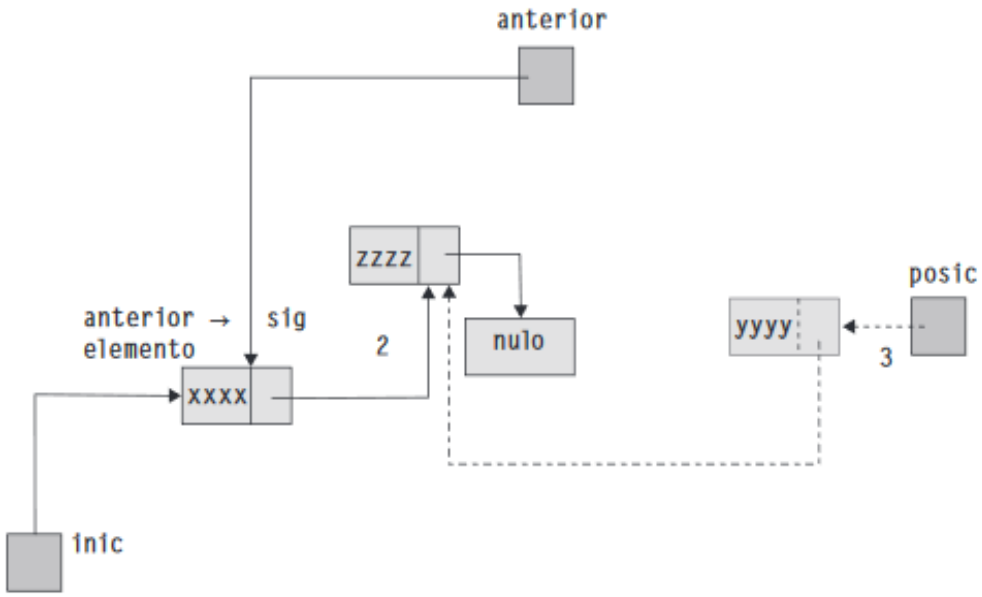
Al suprimir un elemento de una lista consideraremos dos casos particulares:

- El elemento a suprimir está al principio de la lista.
- El elemento se encuentra en cualquier otro lugar de la lista.

1. Situación de partida.
2. `anterior →.sig` apunta a donde `posic →.sig`.
3. `liberar(posic)`.



Cristian Pilaparia 1095211-2202112333




```
procedimiento suprimir(E/S punt: inic, anterior, posic)
inicio
   si anterior = nulo entonces
      inic ← posic→.sig
   si_no
      anterior→.sig ← posic→.sig
   fin_si
   liberar(posic)
   anterior ← nulo // Opcional
   posic ← inic    // Opcional
fin_procedimiento
```

Java y **C#** permiten la creación de listas enlazadas vinculando objetos nodo sin el empleo de punteros, en el enlace se almacena la referencia al siguiente nodo de la lista. En estos lenguajes, aunque la implementación de una lista enlazada resulta similar, hay que tener en cuenta que al crear un objeto nodo se reserva espacio en memoria para él y que este espacio se libera automáticamente, a través de un proceso denominado recolección automática de basura, cuando dicho objeto (nodo) deja de estar referenciado.

Recorrido de una lista enlazada

Para recorrer la lista utilizaremos una variable de tipo puntero auxiliar.

```
procedimiento recorrer(E punt:inic)
   var punt: posic
   inicio
      posic ← inic
      mientras posic <> nulo hacer
         proc_escribir(posic→.elemento)
         posic ← posic→.sig
      fin_mientras
   fin_procedimiento
```

Cristian Pilapanta 1095211-2202112333

EJEMPLO 12.4

Cálculo del número de elementos de una lista enlazada.

```
procedimiento contar(E punt: primero; S entero: n)
   var punt: p
   inicio
      n ← 0      //contador de elementos
      p ← primero
      mientras p <> nulo hacer
         n ← n + 1
         p ← p →.sig
      fin_mientras
   fin_procedimiento
```

Acceso a un elemento de una lista enlazada

La búsqueda de una información en una lista simplemente enlazada sólo puede hacerse mediante un proceso secuencial o recorrido de la lista elemento a elemento, hasta encontrar la información buscada o detectar el final de la lista.

```
procedimiento consultar(E punt: inic; S punt: posic, anterior;
                        E tipo_elemento: elemento; S lógico: encontrado)
   inicio
      encontrado ← falso
      anterior ← nulo
      posic ← inic
      mientras no igual(posic→.elemento, elemento) y (posic <> nulo) hacer
      { igual es una función que compara los elementos que le pasamos como
      parámetros, recurrimos a ella porque, si se tratara de registros,
      compararíamos únicamente la información almacenada en un
      determinado campo }
         si
      fin_mientras
      si igual(posic→.elemento, elemento) entonces
         encontrado ← verdad
      si_no
         encontrado ← falso
      fin_si
   fin_procedimiento
```

EJEMPLO 12.5

Encontrar el nodo de una lista que contiene la información de valor t, suponiendo que la lista almacena datos de tipo entero.

```
procedimiento encontrar(E punt: primeroE entero: t)
   var punt       : p
      entero: n
   inicio
      n ← 0
      p ← primero
```

Cristian Pilapanta 1095211-2202112333

```
    mientras (p→.info <>t) y (p <> nulo) hacer
       n ← n + 1
       p ← p →.sig
    fin_mientras
    si p→.info = t entonces
       escribir('Se encuentra en el nodo ',n,' de la lista')
    si_no
       escribir('No encontrado')
    fin_si
  fin_procedimiento
```

Considere que la información se encuentra almacenada en la lista de forma ordenada, orden creciente y mejore la eficacia del algoritmo anterior.

```
procedimiento encontrar(E punt: primeroE entero: t)
  var punt : p
     entero: n
  inicio
     n ← 0
     p ← primero
     mientras (p →.info <t) y (p <> nulo) hacer
        n ← n + 1
        p ← p →.sig
     fin_mientras
     si p →.info = t entonces
        escribir('Se encuentra en el nodo ',n,' de la lista')
     si_no
        escribir('No encontrado')
     fin_si
  fin_procedimiento
```

12.4.2. Implementación de listas enlazadas con arrays (arreglos)

Las listas enlazadas deberán implementarse de forma dinámica, pero si el lenguaje no lo permite, lo realizaremos a través de arrays (o arreglos), con lo cual impondremos limitaciones en cuanto al número de elementos que podrá contener la lista y estableceremos una ocupación en memoria constante.

Los nodos podrán almacenarse en *arrays* paralelos o *arrays* de registros. Cuando se empleen *arrays* de registros, el valor (dato o información) del nodo se almacenará en un campo y el enlace con el siguiente elemento se almacenará en otro. Otra posible implementación, como ya se ha dicho antes, es con dos arrays: uno para los datos y otro para el enlace.

Un valor de puntero 0, o bien Z, indica el final de la lista.

	ELEMENTO	SIG
1	XXXXXXXXXXXX	2
2	XXXXXXXXXXXX	4
3		
4	XXXXXXXXXXXX	6
5		
6	XXXXXXXXXXXX	0

	ELEMENTO	SIG
1	XXXXXXXXXXXX	2
2	XXXXXXXXXXXX	4
3		
4	XXXXXXXXXXXX	6
5		
6	XXXXXXXXXXXX	0

Cristian Pilapanta 1095211-2202112334

Para definir la lista se debe especificar la variable que apunta al primer nodo (cabecera), que en nuestro caso denominaremos `inic`.

```
inic ← 1
```

Para insertar un nuevo elemento, que siga a `m[1]` y sea seguido por `m[2]`, lo único que se hará es modificar los punteros.

	M	SIG
1	XXXXXXXXXXXXX	3
2	XXXXXXXXXXXXX	4
3	XXXXXXXXXXXXX	2
4	XXXXXXXXXXXXX	6
5		
6	XXXXXXXXXXXXX	0

Como el nuevo elemento se coloca en la primera posición libre deberemos tener un puntero vacío que apunte a dicha primera posición libre; es decir, utilizaremos el array para almacenar dos listas, la lista de elementos y la lista de vacíos; pues es necesario, al comenzar a trabajar, crear la lista de vacíos de la forma que a continuación se expone:

- `vacío` apunta al primer registro libre.
- En el campo `sig` de cada registro se almacena información sobre el siguiente registro disponible.
- Cuando lleguemos al último registro libre, su campo `sig` recibirá el valor `0`, para indicar que ya no quedan más registros disponibles.

Cristian Pilapanta 1095211-2202112334

1095211220211233

		ELEMENTO	SIG
	1		2
	2		3
vacío → 1	3		4
inic → 0	4		5
	5		6
	6		0

Insertar el primer elemento:

	1	XXXXXXXXXXXXX	0
	2		3
vacío → 2	3		4
inic → 1	4		5
	5		6
	6		0

Al implementar una lista a través de arrays necesitaremos los procedimientos:

```
inicializar( ) iniciar( ) consultar( ) insertar( ) suprimir( )
reservar( ) liberar( )
```

y las funciones:

```
vacía( )     llena( )
```

El procedimiento `reservar( )` nos proporcionará la primera posición vacía para almacenar un nuevo elemento y la eliminará de la lista de vacíos, pasando el puntero de vacíos (`vacío`) a la posición siguiente, `vacío` toma el valor del siguiente `vacío` de la lista.

`Liberar( )` inserta un nuevo elemento en la lista de vacíos. Se podrían adoptar otras soluciones, pero nuestro procedimiento liberar insertará el nuevo elemento en la lista de vacíos por delante, sobrescribiendo el campo `m[posic].sig` para que apunte al que antes era el primer `vacío`. El puntero de inicio de los vacíos (`vacío`) lo cambiará al nuevo elemento.

Creación de la lista

Consideraremos el array como si fuera la memoria del ordenador y guardaremos en él dos listas: la lista de elementos y la de vacíos. El primer elemento de la lista está apuntado por `inic` y por `vacío`, el primero de la lista de vacíos:

```
const
   máx = <expresión>
tipo
   registro: tipo_elemento
      ... : ...
      ... : ...
   fin_registro
   registro: tipo_nodo
      tipo_elemento : elemento
      entero        : sig
      { actúa como puntero, almacenando la posición donde se
      encuentra el siguiente elemento de una lista }
   fin_registro
   array[1..máx] de tipo_nodo: arr
var
   entero         : inic,
                    posic,
                    anterior,
                    vacío
   arr            : m
   // m representa la memoria de nuestra computadora
   tipo_elemento  : elemento
   lógico         : encontrado
inicio
   iniciar(m, vacío)
   inicializar(inic)
   ...
fin
```

Al comenzar:

```
procedimiento inicializar(S entero: inic)     //lista de elementos
   inicio
      inic ← 0
   fin_procedimiento
```

`vacío ← 1` indica que el primer registro libre es `m[1]`

`inic ← 0` `inic` señala que no hay elementos en la lista

	elemento	sig
1		2
2		3
3		4
4		5
5		6
6		0

Cristian Pilaparia 1095211-2202112334
1095211220211233

```
procedimiento iniciar(S arr: m; S entero: vacío) //lista de vacíos
   var
      entero: i
   inicio
      vacío ← 1
      desde i ← 1 hasta máx-1 hacer
         m[i].sig ← i+1
   fin_desde
   m[máx].sig ← 0
   //Como ya no hay más posiciones libres a las que apuntar, recibe un 0
   fin_procedimiento
```

Al trabajar de esta manera conseguiremos que la inserción o borrado de un determinado elemento, *n-ésimo*, de la lista no requiera el desplazamiento de otros.

Inserción de un elemento

Al actualizar una lista se pueden presentar dos casos particulares:

- Desbordamiento (*overflow*).
- Subdesbordamiento o desbordamiento negativo (*underflow*).

El desbordamiento se produce cuando la lista está llena y la lista de espacio disponible está vacía.

El subdesbordamiento se produce cuando se tiene una lista vacía y se desea borrar un elemento de la misma. Luego, para poder insertar un nuevo elemento en una lista enlazada, es necesario comprobar que se dispone de espacio libre para ello. Al insertar un nuevo elemento deberemos recurrir al procedimiento `reservar(...)` que nos proporcionará, a través de `auxi`, la primera posición vacía para almacenar en ella el nuevo elemento, eliminando dicha posición de la lista de vacíos.

Por ejemplo, al insertar el primer elemento:

`auxi ← 1`
`vacío ← 2`

	elemento	sig
1	XXXXXXXXXXXXX	~~2~~ 0
2		3 0
3		4
4		5
5		6
6		0

- `vacío` señala que la primera posición libre es la 1 `auxi ← 1`
- El campo `sig` del registro `m[vacío]` proporciona la siguiente posición vacía y `reservar` hará que `vacío` apunte a esta nueva posición `vacío ← 2`

Al insertar un segundo elemento:

- como `vacío` tiene el valor 2 `auxi ← 2`
- `vacío ← m[2].sig`, es decir `vacío ← 3`

`auxi ← 2`
`vacío ← 3`

	elemento	sig
1	XXXXXXXXXXXXX	~~2~~ 0
2	XXXXXXXXXXXXX	~~3~~
3		4
4		5
5		6
6		0

Cristian Pilapanta 1095211-2202112334

```
procedimiento reservar(S entero: auxi; E arr: m; E/S entero: vacío)
inicio
   si vacío = 0 entonces
      // Memoria agotada
      auxi ← 0
   si_no
      auxi ← vacío
      vacío ← m[vacío].sig
   fin_si
fin_procedimiento
```

El procedimiento insertar colocará un nuevo elemento a continuación de anterior, si anterior fuera 0 significa que ha de insertarse al comienzo de la lista.

```
procedimiento insertar (E/S entero: inic, anterior;
                        E tipo_elemento: elemento;
                        E/S arr: m ; E/S entero: vacío)
var
      entero: auxi
inicio
   reservar(auxi, m, vacío)
   si auxi = 0 entonces OVERFLOW
   m[auxi].elemento ← elemento
   si anterior = 0 entonces
      m[auxi].sig ← inic
      inic ← auxi
   si_no
      m[auxi].sig ← m[anterior].sig
      m[anterior].sig ← auxi
   fin_si
   anterior ← auxi  // Opcional
   {Prepara anterior para que, si no especificamos otra cosa, la
   siguiente inserción se realice a continuación de la actual}
fin_procedimiento
```

Consideremos la siguiente situación y analicemos el comportamiento que en ella tendrían los procedimientos `reservar` e `insertar`: *Se desea insertar un nuevo elemento en la lista a continuación del primero y la situación actual, tras sucesivas inserciones y eliminaciones, es como se muestra a continuación*:

vacío ← 3
inic ← 1

1	XXXXXXXXXXXXX	2
2	XXXXXXXXXXXXX	4
3		5
4	XXXXXXXXXXXXX	6
5		0
6	XXXXXXXXXXXXX	0

El nuevo elemento se colocará en el array en la primera posición libre y lo único que se hará es modificar los punteros.

`reservar(...)` proporciona la primera posición libre

Cristian Pilapanta 1095211-2202112334

auxi ← 3

vacío ← 5

1	XXXXXXXXXXXXX	3
2	XXXXXXXXXXXXX	4
3	nuevo_elemento	2
4	XXXXXXXXXXXXX	6
5		0
6	XXXXXXXXXXXXX	0

```
m[3].elemento ← nuevo_elemento
```

como queremos insertar el nuevo elemento a continuación del primero de la lista, su anterior será el apuntado por `inic`:

```
anterior ← 1
m[3].sig ← 2
m[1].sig ← 3
```

Eliminación de un elemento

Para eliminar un elemento de la lista deberemos recurrir al procedimiento `suprimir(...)`, que, a su vez, llamará al procedimiento `liberar(...)`para que inserte el elemento eliminado en la lista de vacíos.

Supongamos que se trata de eliminar el elemento marcado con ********** cuya posición es 3:

```
posic ← 3
```

el elemento anterior al 3 ocupa en el array la posición 2:

```
anterior ← 2
```

y el primer vacío está en 5. Siendo el aspecto actual de la lista el siguiente:

inic ← 1
anterior ← 2
posic ← 3

vacío ← 5

	elemento	sig
1	XXXXXXXXXXXXX	2
2	XXXXXXXXXXXXX	3
3	**********	4
4	XXXXXXXXXXXXX	0
5		6
6		0

Al suprimir el elemento 3 la lista quedaría:

```
m[2].sig ← 4
```

mediante el procedimiento `liberar(...)` incluimos el nuevo `vacío` en la lista de vacíos:

```
m[3].sig ← 5      vacío ← 3
```

como el que se suprime no es el primer elemento de la lista, el valor de `inic` no varía:

```
inic ← 1
```

	elemento	sig
1	XXXXXXXXXXXXX	2
2	XXXXXXXXXXXXX	4
3	**********	5
4	XXXXXXXXXXXXX	0
5		6
6		0

Cristian Pilapaña 1095211-2202112334

```
procedimiento liberar(E entero: posic; E/S arr: m; E/S entero: vacío)
inicio
   m[posic].sig ← vacío
   vacío ← posic
fin_procedimiento
procedimiento suprimir (E/S entero: inic, anterior, posic; E/S arr: m;
                        E/S entero: vacío)
inicio
   si anterior = 0 entonces
      inic ← m[posic].sig
   si_no
      m[anterior].sig ← m[posic].sig
   fin_si
   liberar(posic, m, vacío)
   anterior ← 0  // Opcional
   posic ← inic  // Opcional
   { Las dos últimas instrucciones preparan los punteros para que,
   si no se especifica otra cosa, la próxima eliminación se realice
   por el principio de la lista }
fin_procedimiento
```

Recorrido de una lista

El recorrido de la lista se realizará siguiendo los punteros a partir de su primer elemento, el señalado por `inic`.

El procedimiento `recorrer(...)` que se implementa a continuación, al recorrer la lista va mostrando por pantalla los diferentes elementos que la componen.

```
procedimiento recorrer(E entero: inic)
   var entero: posic
   inicio
      posic ← inic
      mientras posic <> 0 hacer
         { Recurrimos a un procedimiento, proc_escribir(...),
            para presentar por pantalla los campos del registro
            pasado como parámetro }
         proc_escribir(m[posic].elemento)
         posic ← m[posic].sig
      fin_mientras
fin_procedimiento
```

Búsqueda de un determinado elemento en una lista

El procedimiento `consultar` informará sobre si un determinado elemento se encuentra o no en la lista, la posición que ocupa dicho elemento en el array y la que ocupa el elemento anterior. Si la información se encontrara colocada en la lista de forma ordenada y creciente por el campo de búsqueda el procedimiento de consulta podría ser el siguiente:

```
procedimiento consultar (E entero: inic; S entero: posic, anterior;
                         E tipo_elemento: elemento; S lógico: encontrado;
                         E arr: m)
inicio
   anterior ← 0
   posic ← inic
   { Las funciones menor() e igual()comparan los registros por
      un determinado campo}
```

Cristian Pilaparita 1095211-2202112334

```
   mientras menor(m[posic].elemento, elemento) y (posic <> 0) hacer
      anterior ← posic
      posic ← m[posic].sig
   fin_mientras
   si(posic = 0) entonces
      encontrado ← falso
   si igual(m[posic].elemento, elemento) entonces
      encontrado ← verdad
fin_procedimiento
```

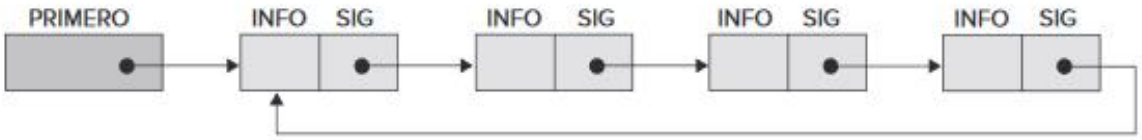
Funciones

Cuando implementamos una lista enlazada utilizando arrays, necesitamos las siguientes funciones:

```
lógico función vacía(E entero: inic)
   inicio
      devolver(inic = 0)
   fin_función
lógico función llena(E entero: vacío)
   inicio
      devolver(vacío = 0)
   fin_función
```

12.5. Listas circulares

Las listas simplemente enlazadas no permiten a partir de un elemento acceder directamente a cualquiera de los elementos que le preceden. En lugar de almacenar un puntero NULO en el campo SIG del último elemento de la lista, se hace que el último elemento apunte al primero o principio de la lista. Este tipo de estructura se llama *lista enlazada circular* o simplemente *lista circular* (en algunos textos se les denomina **listas en anillo**).



Las listas circulares presentan las siguientes *ventajas* respecto de las listas enlazadas simples:

- Cada nodo de una lista circular es accesible desde cualquier otro nodo de ella; es decir, dado un nodo se puede recorrer toda la lista completa. En una lista enlazada de forma simple sólo es posible recorrerla por completo si se parte de su primer nodo.
- Las operaciones de concatenación y división de listas son más eficaces con listas circulares.

Los *inconvenientes*, por el contrario, son:

- Se pueden producir lazos o bucles infinitos. Una forma de evitar estos bucles infinitos es disponer de un nodo especial que se encuentre permanentemente asociado a la existencia de la lista circular. Este nodo se denomina *cabecera* de la lista.

Figura 12.7. Nodo cabecera de la lista.

Cristian Pilaparia 1095211-2202112334

El nodo cabecera puede diferenciarse de los otros nodos en una de las dos formas siguientes:

- Puede tener un valor especial en su campo INFO que no es válido como datos de otros elementos.
- Puede tener un indicador o bandera *(flag)* que señale cuando es nodo cabecera.

El campo de la información del nodo cabecera no se utiliza, lo que se señala con el sombreado de dicho campo.

Una lista enlazada circularmente *vacía* se representa como se muestra en la figura 12.8.

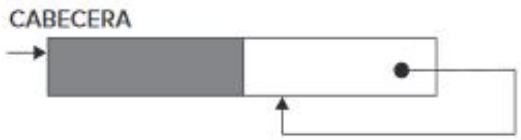


Figura 12.8. Lista circular vacía.

12.6. Listas doblemente enlazadas

En las listas lineales estudiadas anteriormente el recorrido de ellas sólo podía hacerse en un único sentido: *de izquierda a derecha* (principio a final). En numerosas ocasiones se necesita recorrer las listas en ambas direcciones, cuando esto es posible, se denominan *listas doblemente enlazadas*. En estas listas cada nodo consta del campo INFO de datos y dos campos de enlace o punteros: ANTERIOR(ANT) y SIGUIENTE(SIG) que apuntan hacia adelante y hacia atrás (figura 12.9). Como cada elemento tiene dos punteros, una lista doblemente enlazada ocupa más espacio en memoria que una lista simplemente enlazada para una misma cantidad de información.

La lista necesita dos punteros, CABECERA y FIN,[2] que apuntan hacia el primero y último nodo.

La variable CABECERA y el puntero SIG permiten recorrer la lista en el sentido normal y la variable FIN y el puntero ANT permiten recorrerla en sentido inverso.

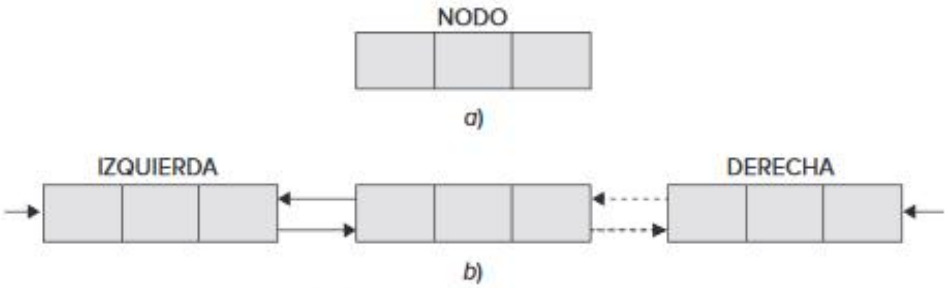


Figura 12.9. Lista doblemente enlazada.

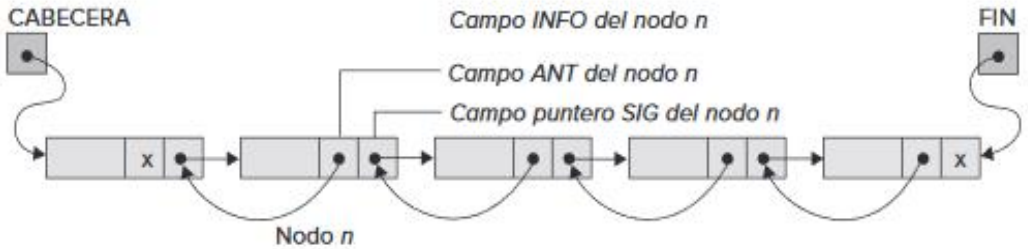


Figura 12.10. Lista doble.

[2] Se adoptan estos términos a efectos de normalización, pero el lector puede utilizar IZQUIERDA y DERECHA.

Cristian Pilaparia 1095211-2202112334

Como se ve en la figura 12.11, una propiedad fundamental de las listas doblemente enlazadas es que para cualquier puntero P de la lista:

```
nodo [nodo[p].sig].ant = p
nodo [nodo[p].ant].sig = p
```

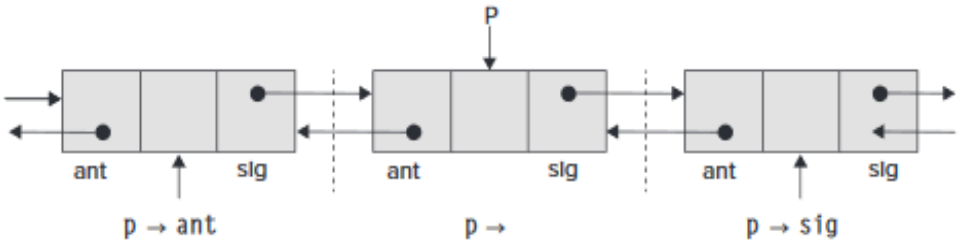


Figura 12.11.

12.6.1. Inserción

La inserción de un nodo a la derecha de uno especificado, cuya dirección está dada por la variable M, puede presentar varios casos:

1. La lista está vacía, se indica mediante M = NULO, CABECERA y FIN son también NULO. Una inserción indica que CABECERA se debe fijar con la dirección del nuevo nodo y los campos ANT y SIG también se establecen en NULO.
2. Insertar dentro de la lista: existe un elemento anterior y otro posterior de nuevo nodo.
3. Insertar a la derecha del nodo del fin de la lista, se requiere que el apuntador FIN sea modificado.

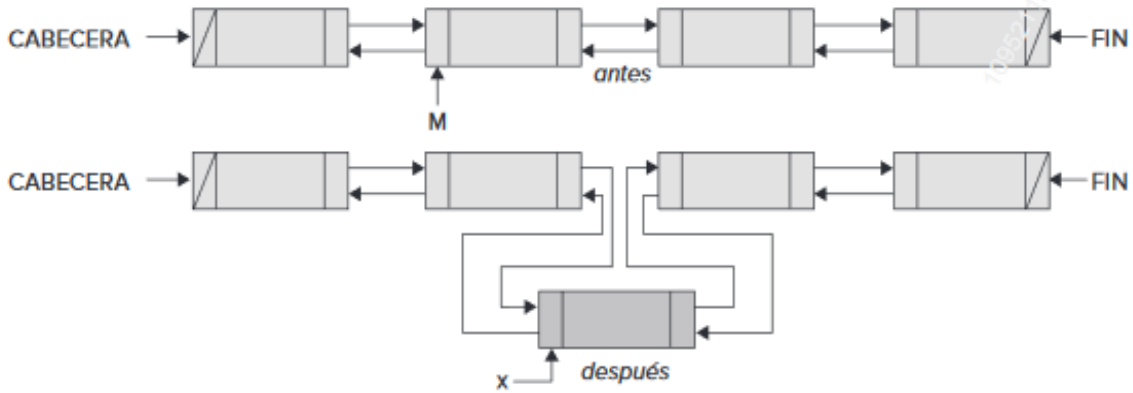


Figura 12.12. Inserción en una lista doblemente enlazada.

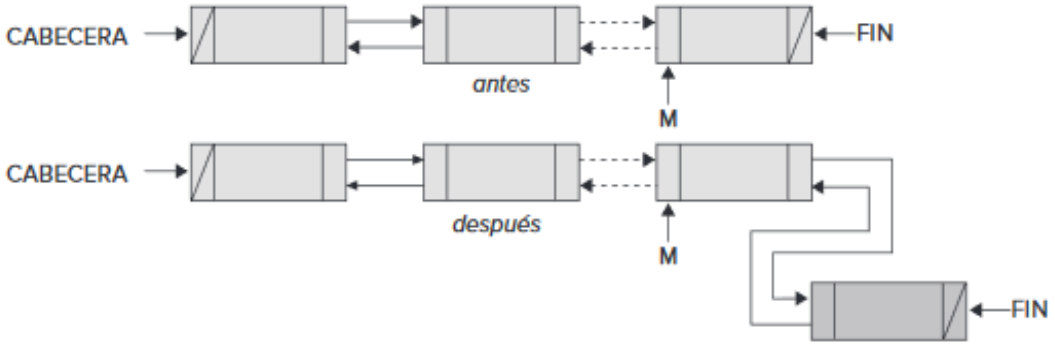


Figura 12.13. Inserción en el extremo derecho de una lista doblemente enlazada.

Cristian Pilapanta 1095211-2202112334

12.6.2. Eliminación

La operación de eliminación es directa. Si la lista tiene un simple nodo, entonces los punteros de los extremos izquierdo y derecho asociados con la lista se deben fijar en `NULO`. Si el nodo del extremo derecho de la lista es el señalado para la eliminación, la variable `FIN` debe modificarse para señalar el predecesor del nodo que se va a borrar de la lista. Si el nodo del extremo izquierdo de la lista es el que se desea borrar, la variable `CABECERA` debe modificarse para señalar el elemento siguiente.

La eliminación se puede realizar dentro de la lista (figura 12.14).

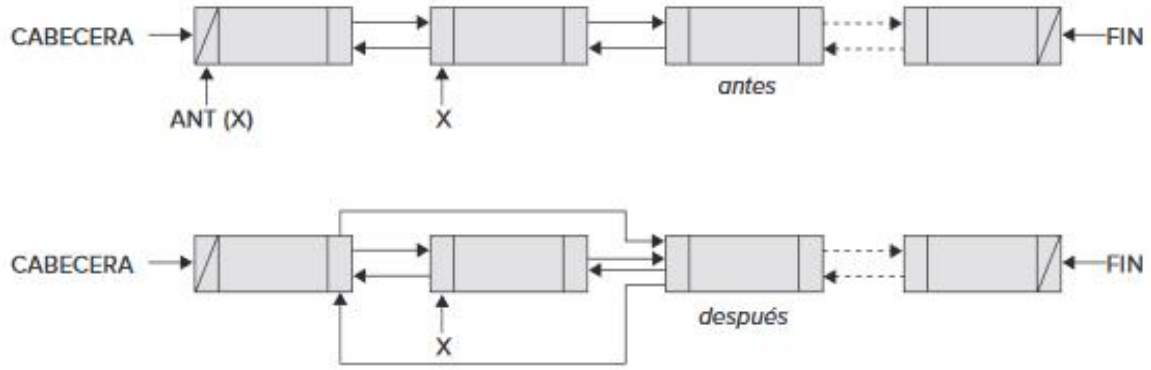


Figura 12.14. Eliminación de un nodo X en una lista doblemente enlazada.

12.7. Pilas

Una *pila (stack)* es un tipo especial de lista lineal en la que la inserción y borrado de nuevos elementos se realiza sólo por un extremo que se denomina *cima* o *tope (top)*.

La pila es una estructura con numerosas analogías en la vida real: una pila de platos, una pila de monedas, una pila de cajas de zapatos, una pila de camisas, una pila de bandejas, etcétera.

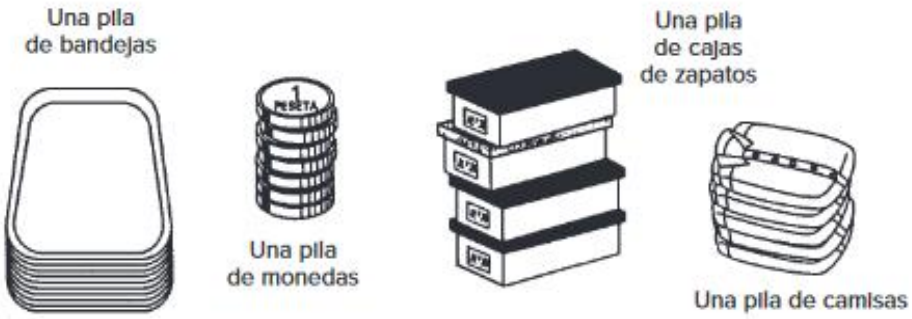


Figura 12.15. Ejemplos de tipos de pilas.

Dado que las operaciones insertar y eliminar se realizan por un solo extremo (el superior), los elementos sólo pueden eliminarse en orden inverso al que se insertan en la pila. El último elemento que se pone en la pila es el primero que se puede sacar; por ello, a estas estructuras se les conoce por el nombre de **LIFO** (*last-in, first-out*, último en entrar, primero en salir).

Las operaciones más usuales asociadas con las pilas son:

`"push"`	*Meter, poner o apilar:* operación de insertar un elemento en la pila.
`"pop"`	*Sacar, quitar o desapilar:* operación de eliminar un elemento de la pila.

Cristian Pilapanta 1095211-2202112334

Las pilas se pueden representar en cualquiera de las tres formas de la figura 12.16.

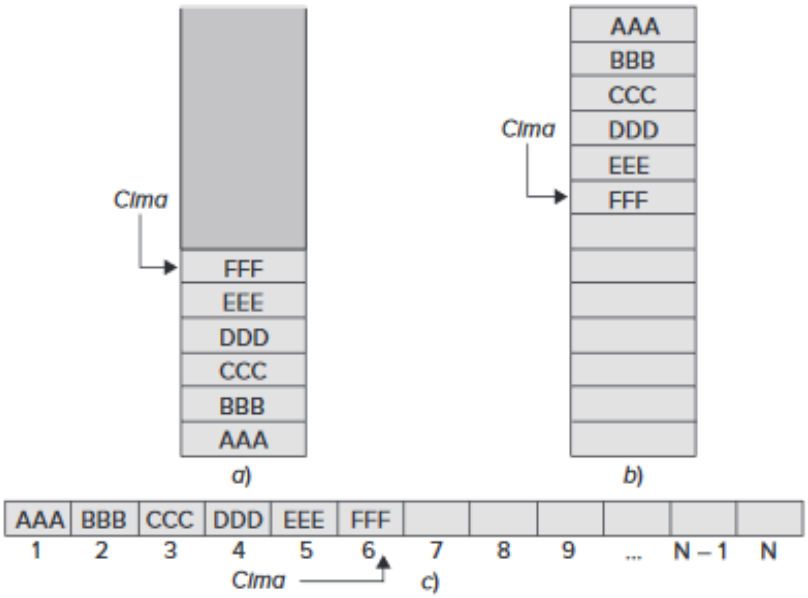


Figura 12.16. Representación de las pilas.

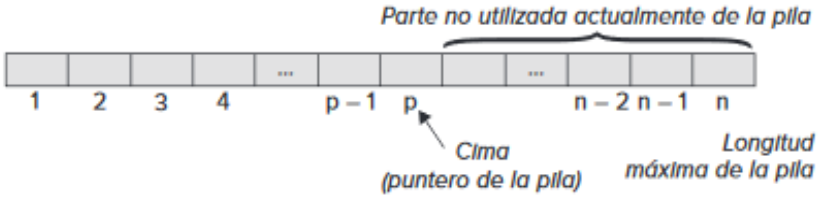
Para representar una pila St, se debe definir un vector con un determinado tamaño (longitud máxima):

```
var array [1..n] de <tipo_dato> : St
```

Se considerará un elemento entero P como el puntero de la pila (*stack pointer*). P es el subíndice del array correspondiente al elemento cima de la pila (esto es, el que ocupa la última posición). Si la pila está vacía, P = 0 (vea la figura de la página siguiente).

En principio, la pila está vacía y el puntero de la pila o CIMA está a cero. Al meter un elemento en la pila, se incrementa el puntero en una unidad; al sacar un elemento de la pila se decrementa en una unidad.

Al manipular una pila se deben realizar algunas comprobaciones; en una pila vacía no se pueden sacar datos (P = 0), si la pila se implementa con un array de tamaño fijo, se puede llenar cuando P = n (n, longitud total de la pila) y el intento de introducir más elementos en la pila producirá un *desbordamiento de la pila*.



Idealmente una pila puede contener un número ilimitado de elementos y no producir nunca desbordamiento. En la práctica, sin embargo, el espacio de almacenamiento disponible es finito. La codificación de una pila requiere un cierto equilibrio, ya que si la longitud máxima de la pila es demasiado grande se gasta mucha memoria, mientras que un valor pequeño de la longitud máxima producirá desbordamientos frecuentes.

Para trabajar fácilmente con pilas es conveniente diseñar subprogramas de *poner (push)* y *quitar (pop)* elementos. También es necesario con frecuencia comprobar si la pila está vacía; esto puede conseguirse con una variable o función booleana VACÍA, de modo que cuando su valor sea *verdadero* la pila está vacía y *falso* en caso contrario.

P = CIMA	*Puntero de la pila.*
VACÍA	*Función booleana "pila vacía".*

Cristian Pilapanta 1095211-2202112334

PUSH	*Subprograma para añadir, poner o insertar elementos.*
POP	*Subprograma para eliminar o quitar elementos.*
LONGMÁX	*Longitud máxima de la pila.*
X	*Elemento a añadir/quitar de la pila.*

Implementación con punteros

Si el lenguaje tiene punteros, deberemos implementar las pilas con punteros.

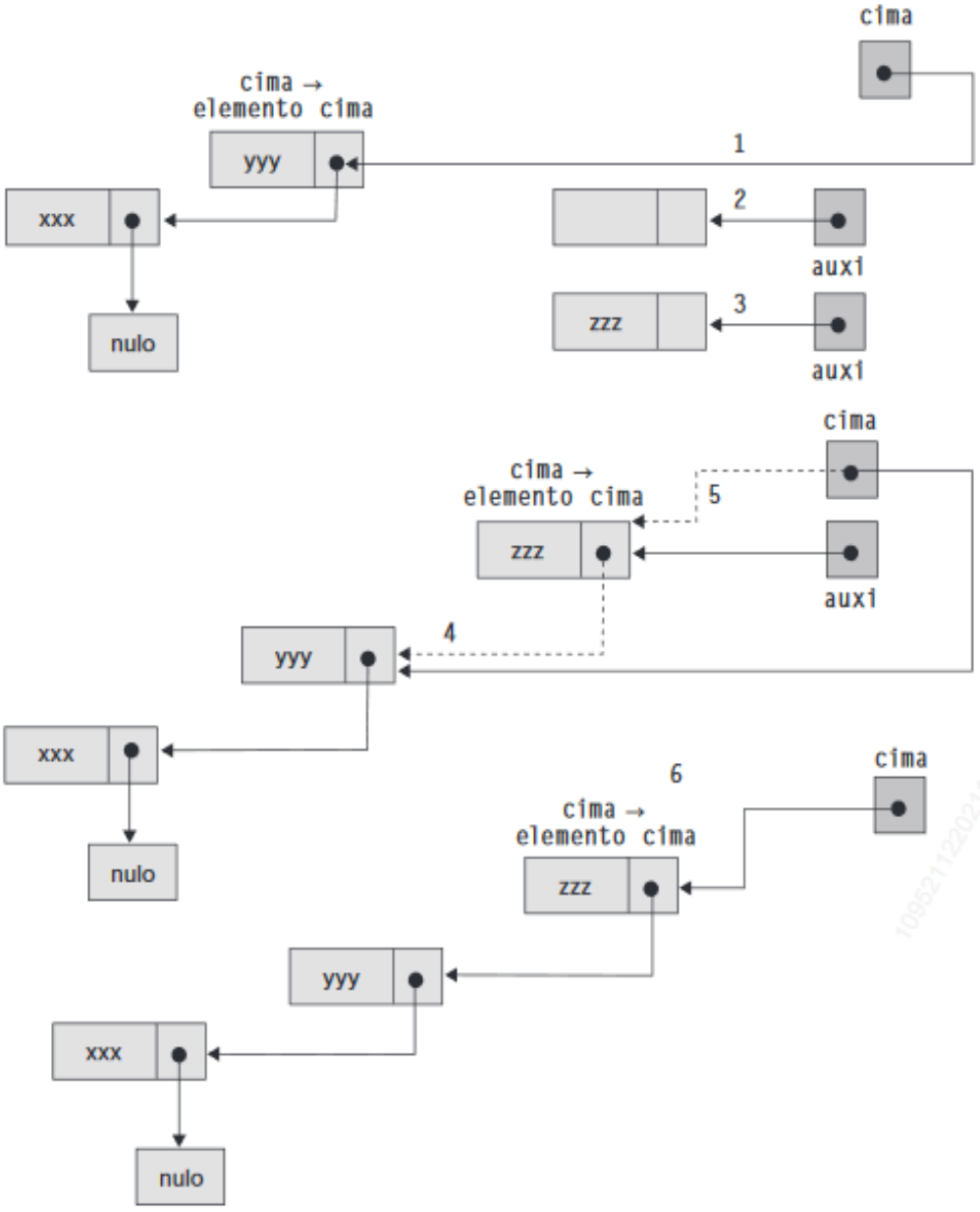
Para la manipulación de una pila mediante punteros es preciso diseñar los siguientes procedimientos y/o funciones: `inicializar` o `crear`, `apilar` o `meter`, `desapilar` o `sacar`, `consultarCima` y `Vacía`.

```
algoritmo pilas_con_punteros
   tipo
      puntero_a nodo: punt
      registro : tipo_elemento
         .... : ....
         .... : ....
      fin_registro
      registro : nodo
         tipo_elemento : elemento
         punt          : cima
   fin_registro
var
   punt     : cima
   elemento : tipo_elemento

   inicio
      inicializar(cima)
      ...
   fin

   procedimiento inicializar(S punt: cima)
      inicio
         cima ← nulo
      fin_procedimiento
   lógico función vacía(E punt: cima)
      inicio
         devolver (cima = nulo)
      fin_función

procedimiento consultarCima(E punt: cima;
                            S tipo_elemento: elemento)
   inicio
   si no vacía (cima) entonces
      elemento ← cima→.elemento
   fin_si
   fin_procedimiento
```

Los elementos se incorporan siempre por un extremo, `cima`.

```
meter(cima, elemento)
```

Cristian Pilaparita 1095211-2202112334



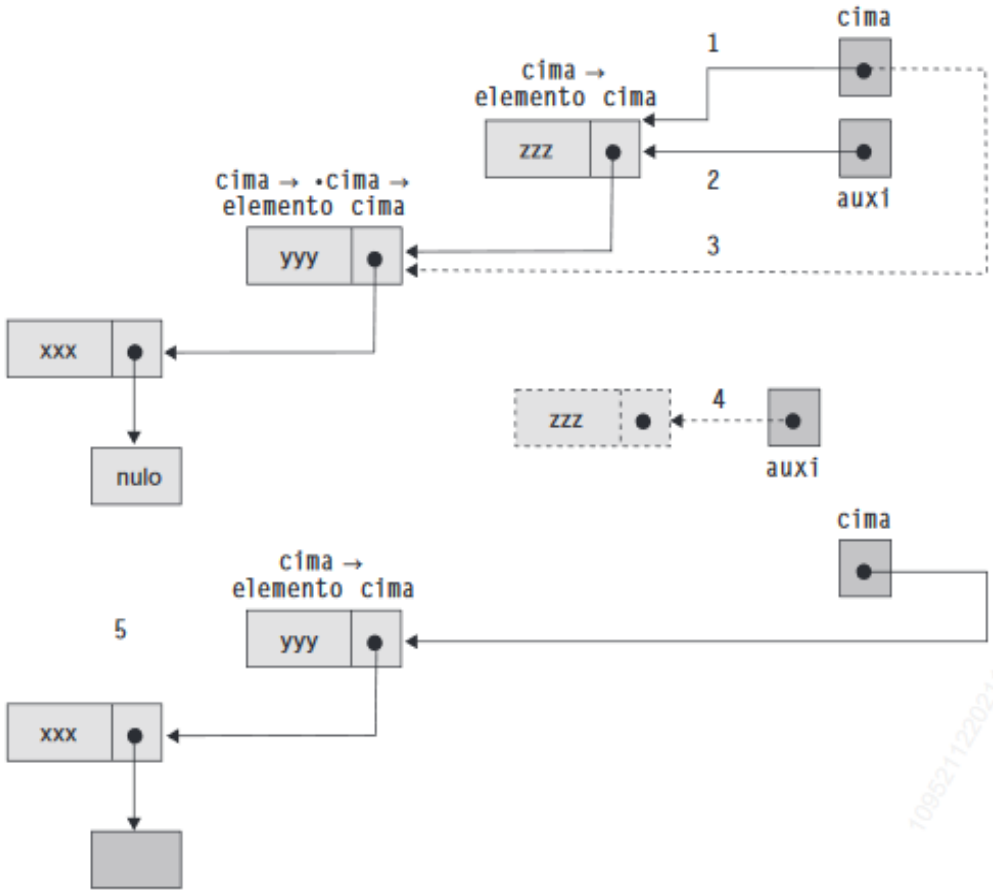
1. `cima` apunta al último elemento de la pila.
2. `reservar(auxi)`.
3. Introducimos la información en `auxi→.elemento`.
4. Hacemos que `auxi→.cima` apunte a donde `cima`.
5. Cambiamos `cima` para que apunte donde `auxi`.
6. La pila tiene un elemento más.

```
procedimiento meter(E/S punt: cima; E tipo_elemento: elemento)
   var
      punt: auxi
   inicio
      reservar(auxi)
      auxi→.elemento ← elemento
      auxi→.cima ← cima
      cima ← auxi
   fin_procedimiento
```

Cristian Pilaparita 1095211-2202112334

Los elementos se recuperan en orden inverso a como fueron introducidos:

```
Sacar(cima, elemento)
```



1. = cima apunta al último elemento de la pila.
2. Hacemos que auxi apunte adonde apuntaba cima.
3. Y que cima pase a apuntar adonde cima→.cima.
4. liberar(auxi).
5. La pila tiene un elemento menos.

```
procedimiento sacar(E/S punt:cima; S tipo_elemento: elemento)
   var
      punt: auxi
   inicio
      si no vacía (cima) entonces
         auxi ← cima
         elemento ← cima →.elemento
         cima ← cima →.cima
      liberar(auxi)
      {liberar es un procedimiento para la eliminación de
      variables dinámicas}
   fin_si
fin_procedimiento
```

Cristian Pilaparia 1095211-2202112334

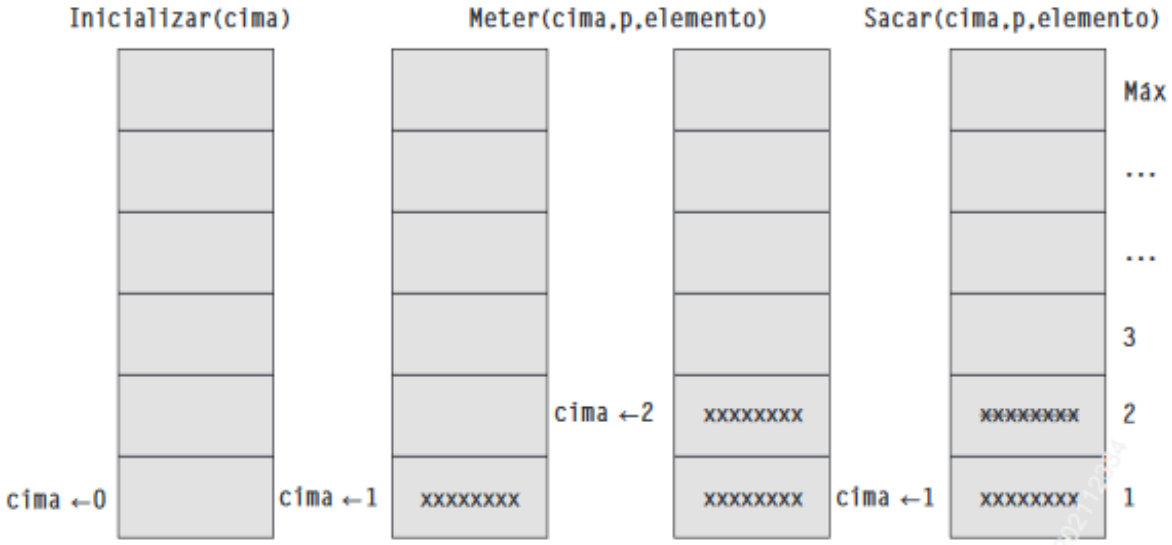
Implementación con arrays

Necesitaremos un array y una variable numérica `cima` que apunte al último elemento colocado en la pila.

La inserción o extracción de un elemento se realizará siempre por la parte superior.

Su implementación mediante arrays limita el máximo número de elementos que la pila puede contener y origina la necesidad de una función más.

`Llena(...)` de resultado lógico




```
const Máx = <expresión>
tipo
   registro: tipo_elemento
      ... : ...
      ... : ...
   fin_registro
   array[1..Máx] de tipo_elemento: arr
var
   entero        : cima
   arr           : p
   tipo_elemento : elemento
   inicio
      inicializar(cima)
      ...
   fin

   procedimiento inicializar(S entero: cima)
      inicio
         cima ← 0
      fin_procedimiento

   lógico función vacía(E entero: cima)
      inicio
         si cima = 0 entonces
```

Cristian Pilapanta 1095211-2202112334

```
      devolver (verdad)
   si_no
      devolver (falso)
   fin_si
fin_función

lógico función llena(E entero: cima)
   inicio
   si cima = Máx entonces
      devolver (verdad)
   si_no
      devolver (falso)
   fin_si
fin_función

procedimiento consultarCima(E entero: cima; E arr:p;
                            S tipo_elemento: elemento)
   inicio
   si no vacía (cima) entonces
      elemento ← p[cima]
   fin_si
   fin_procedimiento

procedimiento meter(E/S entero: cima; E/S arr: p;
                    E tipo_elemento: elemento)
   inicio
   si no llena (cima) entonces
      cima ← cima + 1
      p[cima] ← elemento
   fin_si
fin_procedimiento

procedimiento sacar(E/S entero: cima; E arr: p;
                    S tipo_elemento: elemento)
inicio
si no vacía (cima) entonces
   elemento ← p[cima]
   cima ← cima - 1
fin_si
fin_procedimiento
```

12.7.1. Aplicaciones de las pilas

Las pilas son utilizadas ampliamente para solucionar una gran variedad de problemas; se utilizan en compiladores, sistemas operativos y en programas de aplicación. Veamos algunas de las aplicaciones más interesantes.

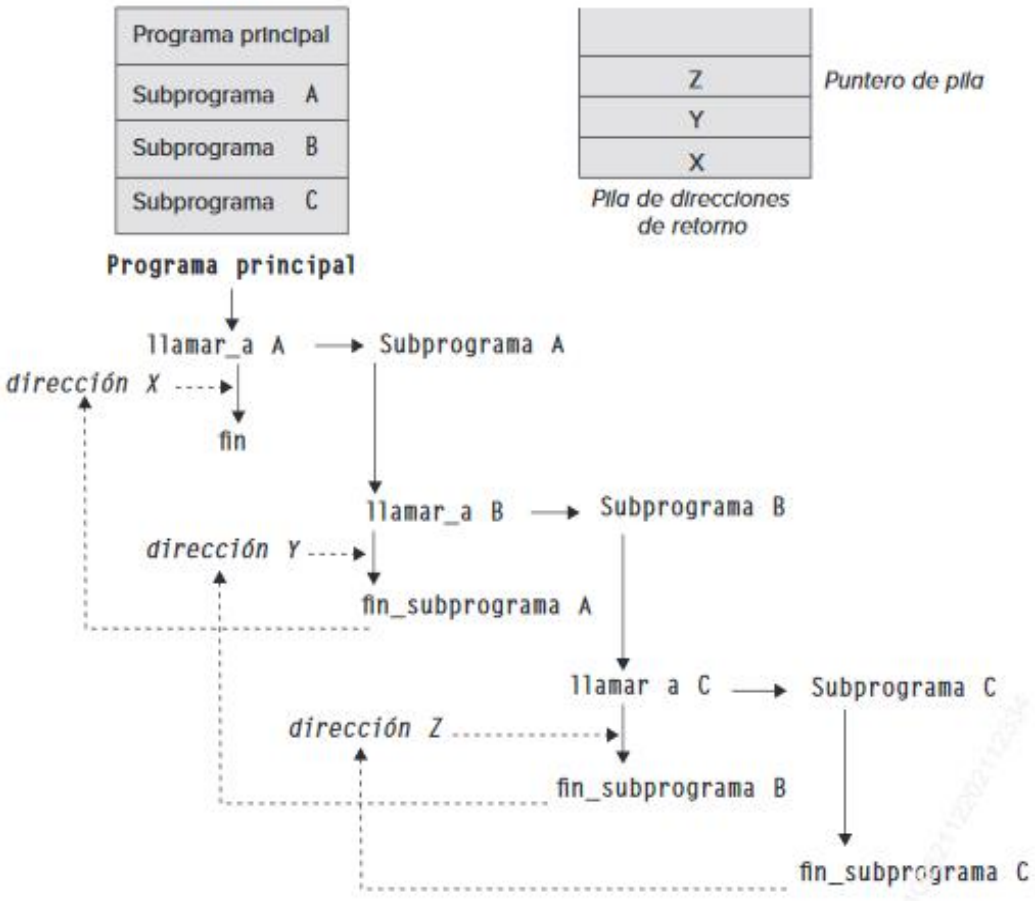
Llamadas a subprogramas

Cuando dentro de un programa se realizan llamadas a subprogramas, el programa principal debe recordar el lugar donde se hizo la llamada, de modo que pueda retornar allí cuando el subprograma se haya terminado de ejecutar.

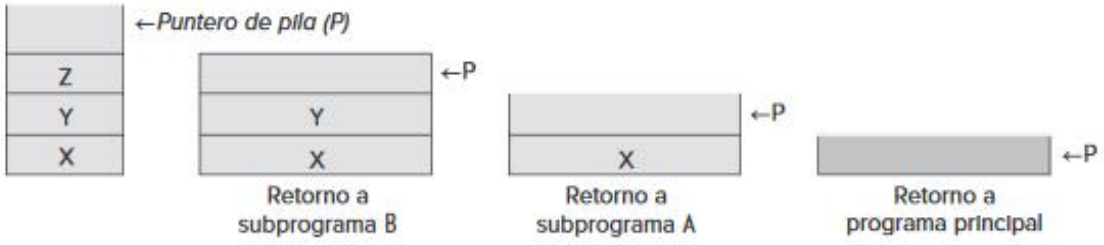
Supongamos que tenemos tres subprogramas llamados A, B y C, y también que A invoca a B y B invoca a C. Entonces B no terminará su trabajo hasta que C haya terminado y devuelto su control a B. De modo similar, A es el primero que arranca su ejecución, pero es el último que la termina, tras la terminación y retorno de B.

Cristian Pilapanta 1095211-2202112334

Esta operación se consigue disponiendo las direcciones de retorno en una pila.



Cuando un subprograma termina, debe retornar a la dirección siguiente a la instrucción que le llamó (**llamar_a**). Cada vez que se invoca un subprograma, la dirección siguiente (X, Y o Z) se introduce en la pila. El vaciado de la pila se realizará por los sucesivos retornos, decrementándose el puntero de pila que queda libre apuntando a la siguiente dirección de retorno.



EJEMPLO 12.6

Se desea leer un texto y separar los caracteres letras, dígitos y restantes caracteres para ser utilizados posteriormente.

Utilizaremos tres pilas (LETRAS, DÍGITOS, OTROSCAR) para contener los diferentes tipos de caracteres. El proceso consiste en leer carácter a carácter, comprobar el tipo de carácter y según el resultado introducirlo en su pila respectiva.

```
algoritmo lecturacarácter
   const Máx = <valor>
tipo
```

Cristian Pilapanta 1095211-2202112334

```
   array [1..Máx] de carácter:pila
var
   entero         : cima1, cima2, cima3
   pila           : pilaletras, piladígitos, pilaotroscaracteres
   carácter       : elemento
inicio
   crear (cima1)
   crear (cima2)
   crear (cima3)
   elemento ← leercar
   mientras (código(elemento)<> 26) y no llena(cima1) y no
            llena(cima2)y no llena(cima3) hacer
      {saldremos del bucle en cuanto se llene alguna de las pilas o
            pulsemos ^Z}
      si (elemento >= 'A') y (elemento <= 'Z') o (elemento >= 'a') y
            (elemento >= 'z')entonces
         meter (cima1, pilaletras, elemento)
      si_no
         si (elemento>= '0') y (elemento<='9') entonces
            meter (cima2, piladígitos, elemento)
         si_no
            meter (cima3, pilaotroscaracteres, elemento)
         fin_si
      fin_si
      elemento ← leercar
   fin_mientras
fin

procedimiento crear (S entero: cima)
   inicio
     cima ← 0
   fin_procedimiento

lógico función llena (E entero: cima)
   inicio
      devolver (cima = Máx)
   fin_función

procedimiento meter(E/S entero: cima; E/S tipo_elemento: elemento)
   inicio
      cima ← cima+1
      p [cima] ← elemento
   fin_procedimiento
```

12.8. Colas

Las colas son otro tipo de estructura lineal de datos similar a las pilas, diferenciándose de ellas en el modo de insertar/eliminar elementos.

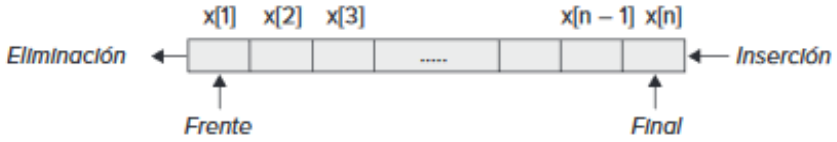
Una *cola (queue)* es una estructura lineal de datos:

```
var array [1..n] de <tipo_dato> : C
```

en la que las *eliminaciones* se realizan al principio de la lista, *frente (front)*, y las *inserciones* se realizan en el otro extremo, *final (rear)*. En las colas, el primer elemento que entra, es también el primero que sale; por ello se conoce como listas **FIFO** (*first-in, first-out*, "primero en entrar, primero en salir"). Así, pues, la diferencia con las pilas resi-

Cristian Pilaparita 1095211-2202112334
1095211220211233

de en el modo de entrada/salida de datos; en las colas las inserciones se realizan al final de la lista, no al principio. Por ello las colas se usan para almacenar datos que necesitan ser procesados según el orden de llegada.

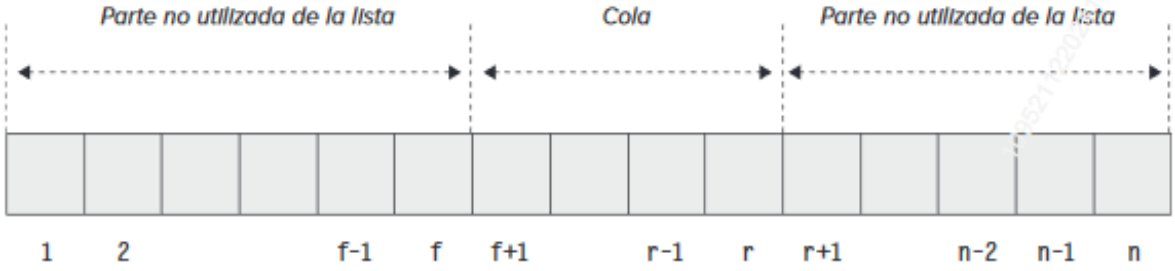


En la vida real se tienen diversos ejemplos de colas: la de un autobús, la de un cine, una caravana de coches en una calle, entre otras. En todas ellas el primer elemento (pasajero, coche, etcétera) que llega es el primero que sale.

En informática existen también numerosas aplicaciones de las colas. Por ejemplo, en un sistema de tiempo compartido suele haber un procesador central y una serie de periféricos compartidos: discos, impresoras, etcétera. Los recursos se comparten por los diferentes usuarios y se utiliza una cola para almacenar los programas o peticiones de los diferentes usuarios que esperan su turno de ejecución. El procesador central atiende —normalmente— por riguroso orden de llamada del usuario; por tanto, todas las llamadas se almacenan en una cola. Existe otra aplicación muy utilizada que se denomina *cola de prioridades*; en ella el procesador central no atiende por riguroso orden de llamada, aquí el procesador atiende por prioridades asignadas por el sistema o bien por el usuario, y sólo dentro de las peticiones de igual prioridad se producirá una cola.

12.8.1. Representación de las colas

Las colas se pueden representar por listas enlazadas o por arrays. Se necesitan dos punteros: *frente(f)* y *final(r)*, y la lista o array de *n* elementos (`LONGMÁX`).



si la cola está vacía	frentre = nulo	o bien	f ← 0
eliminar elementos	frentre ← frentre + 1	o bien	f ← f + 1
añadir elementos	final ← final + 1	o bien	f ← r + 1

La figura 12.17 muestra la representación de una cola mediante un array o mediante una lista enlazada. Las operaciones que se pueden realizar con una cola son:

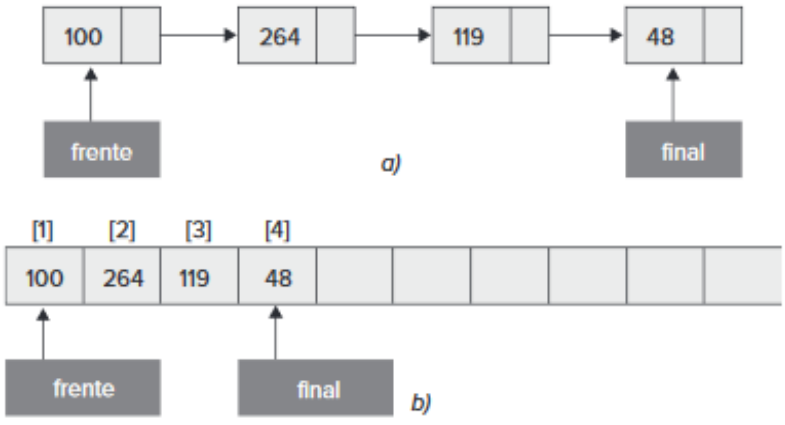


Figura 12.17. Representación de una cola: *a*) mediante una lista enlazada; *b*) mediante un array.

Cristian Pilapanta 1095211-2202112334

- Acceder al primer elemento de la cola.
- Añadir un elemento al final de cola.
- Eliminar el primer elemento de la cola.
- Vaciar la cola.
- Verificar el estado de la cola: vacía o llena.

Implementación con estructuras dinámicas

Aunque, como posteriormente veremos, las colas se pueden simular mediante un array y dos variables numéricas (`frente, final`), si el lenguaje lo permite deberemos implementarlas mediante punteros.

En una cola las eliminaciones se realizarán por el extremo denominado `frente` y las inserciones por el `final`. Para la manipulación de una cola necesitaremos los subprogramas: `inicializar` o `crear`, `consultarPrimero`, `poner` o `meter`, `quitar` o `sacar` y `vacía` o `colaVacía` y los siguientes tipos de datos:

```
tipo
 puntero_a nodo : punt
 registro       : tipo_elemento
    ... : ....
    ... : ....
 fin_registro
 registro         : nodo
    tipo_elemento : elemento
    punt          : sig
 fin_registro
var
 punt          : frente, final
 tipo_elemento : elemento
```

Cuando no hay elementos en la cola:

```
frente ← nulo    final ← nulo
```

de lo que deducimos:

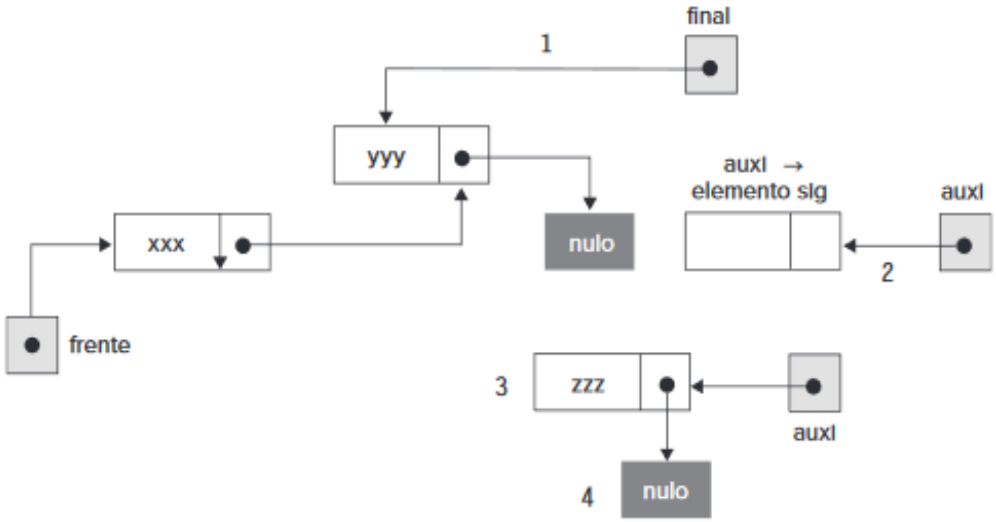
```
procedimiento inicializar(S punt: frente, final)
 inicio
  frente ← nulo
  final ← nulo
 fin_procedimiento
```

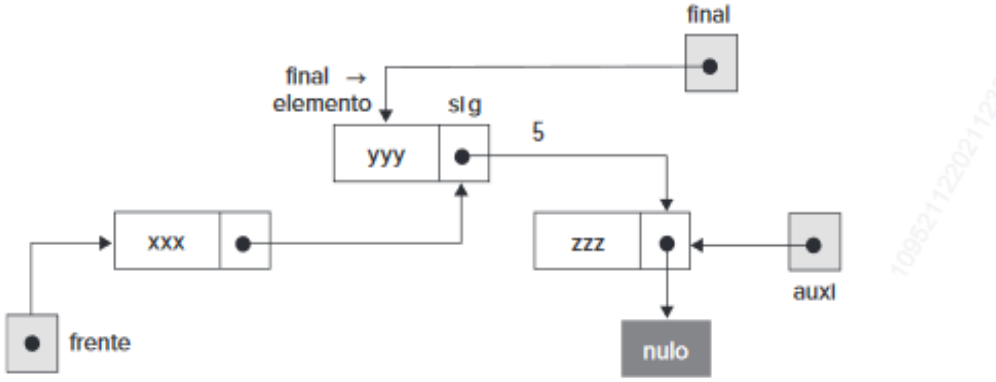
```
Meter(final, frente, elemento)
```

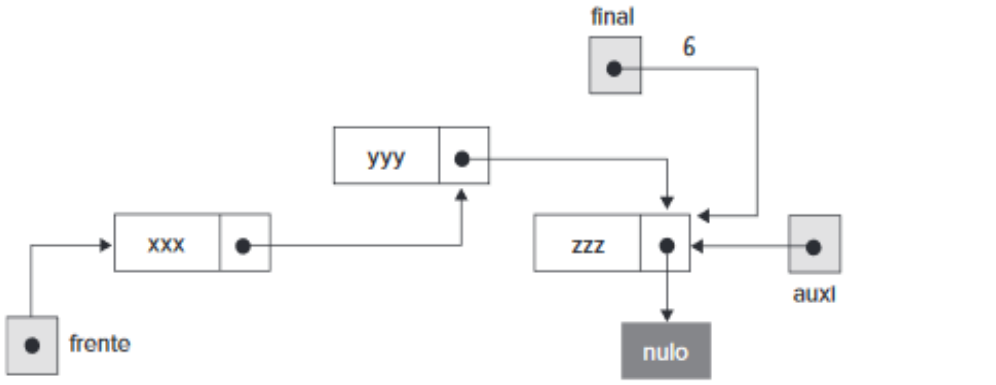
```
procedimiento meter(E/S punt: final; S punt: frente;
                    E tipo_elemento: elemento)
   var
     punt: auxi
   inicio
    reservar(auxi)
    auxi→.elemento ← elemento
    auxi→.sig      ← nulo
    si final = nulo entonces
       frente ← auxi
    si_no
       final→.sig ← auxi
    fin_si
    final ← auxi
   fin_procedimiento
```

Cristian Pilaparita 1095211-2202112335

```
Sacar(frente,final,elemento)
```







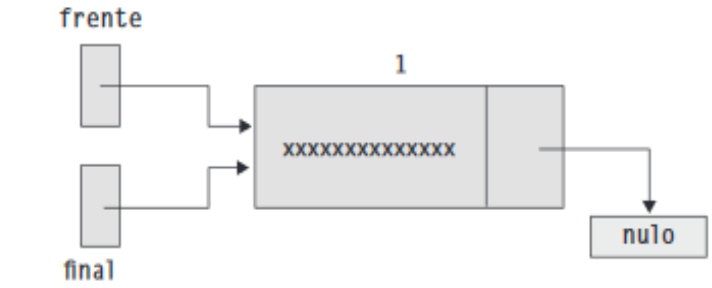
Los pasos de la operación `meter ()`:

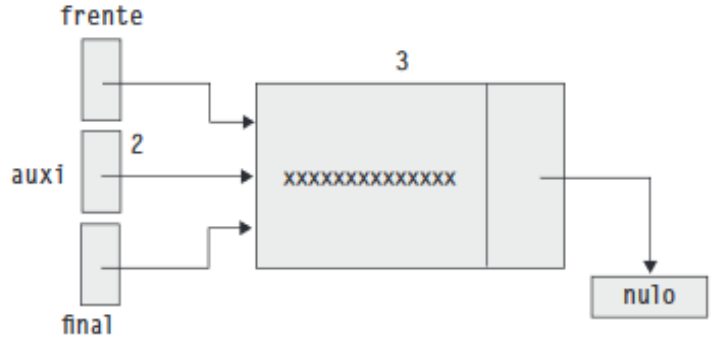
1. Situación de partida.
2. `reservar (aux1)`.
3. Introducir la nueva información en `aux1 →.elemento`.
4. Hacer que `aux1 →.sig` apunte a `nulo`.

Cristian Pilapanta 1095211-2202112334

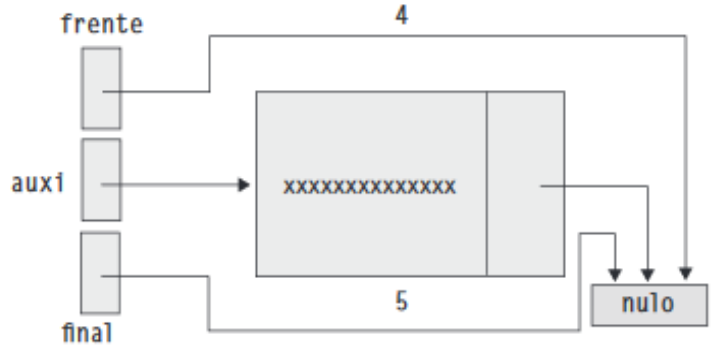
5. Conseguir que `final →.sig` apunte a donde lo hace `auxi`.
6. Por último, `final` debe apuntar también a donde `auxi`.

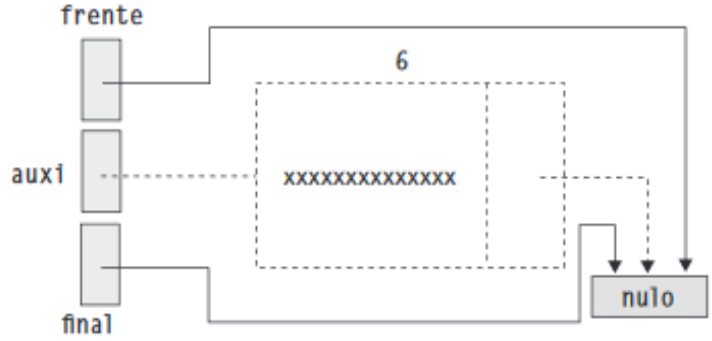
Imaginemos que la cola tiene un único elemento. Los elementos se extraen siempre por el `frente`.





1. Estado inicial.
2. Hacemos que `auxi` apunte donde lo hace `frente`.
3. Extraemos la información de `auxi →.elemento`.





Cristian Pilaparia 1095211-2202112335

4. Hacemos que frente apunte a donde lo hace auxi→.sig.
5. Como frente toma el valor nulo a final le damos nulo.
6. Liberar(auxi).

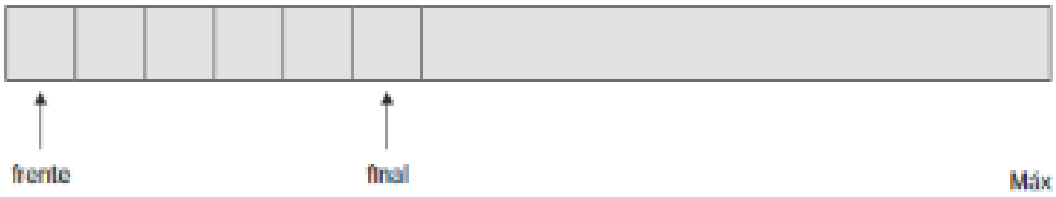
```
procedimiento sacar(E/S punt: frente; S punt: final;
                    S tipo_elemento: elemento)
   var
     punt: auxi
   inicio
    auxi     ← frente
    elemento ← auxi→.elemento
    frente   ← frente→.sig
    si frente = nulo entonces
       final ← nulo
    fin_si
    liberar(auxi)
   fin_procedimiento
```

Como los elementos se extraen siempre por el frente, la cola estará vacía cuando:

```
frente = nulo

  lógico función vacía(E punt: frente)
   inicio
    si frente = nulo entonces
       devolver(verdad)
    si_no
       devolver(falso)
    fin_si
   fin_función

  procedimiento consultarPrimero(E punt: frente; S tipo_elemento: elemento)
  inicio
    si no vacía (frente) entonces
      elemento ← frente→.elemento
    fin_sin
fin_procedimiento
```

Implementación utilizando estructuras de tipo array

Podemos representar las colas mediante arrays.




```
const
   Máx  = <expresión>  //longitud máxima
tipo
   registro: tipo_elemento
      ... : ...
      ... : ...
```

Cristian Paparini 10952112202112355

```
    fin_registro
    array[1..Máx] de tipo_elemento: arr
var
 entero          : frente, final
 arr             : c          //la cola se define como un array
 tipo_elemento   : elemento
```

Cuando la cola esté vacía `frente ← 0` `final ← 0`

```
procedimiento inicializar(S entero: frente,final)
 inicio
  frente ← 0
  final ← 0
 fin_procedimiento
```

El procedimiento para la inserción de un nuevo elemento deberá verificar, en primer lugar, que la cola no está totalmente llena y, por consiguiente, no se producirá error de desbordamiento. La condición de desbordamiento se produce cuando `final = Máx`.

```
procedimiento meter(E/S entero: final; S entero: frente;
                    E/S     arr: c; E tipo_elemento: elemento)
 inicio
  si final = 0 entonces
     frente ← 1
  fin_si
  final ← final + 1
  c[final] ← elemento
 fin_procedimiento
```

Para eliminar un elemento será preciso verificar, en primer lugar, que la cola no está vacía.

```
procedimiento sacar(E/S entero: frente,final; E arr: c;
                    S tipo_elemento: elemento)
  inicio
     si no vacía (frente) entonces
      elemento ← c[frente]
      frente ← frente + 1
      si frente = final + 1 entonces
         frente ← 0
         final ← 0
      fin_si
     fin_si
  fin_procedimiento
```

La cola estará vacía cuando `frente = 0`.

```
 lógico función vacía(E entero: frente)
  inicio
    devolver(frente = 0)
  fin_función
```

Esta implementación tiene el inconveniente de que puede ocurrir que la variable `final` llegue al valor máximo de la tabla, con lo cual no se puedan seguir añadiendo elementos a la cola, aun cuando queden posiciones libres a la izquierda de la posición frente por haber sido eliminados algunos de sus elementos.

Cristian Pilapanta 1095211-2202112335

Existen diversas soluciones a este problema:

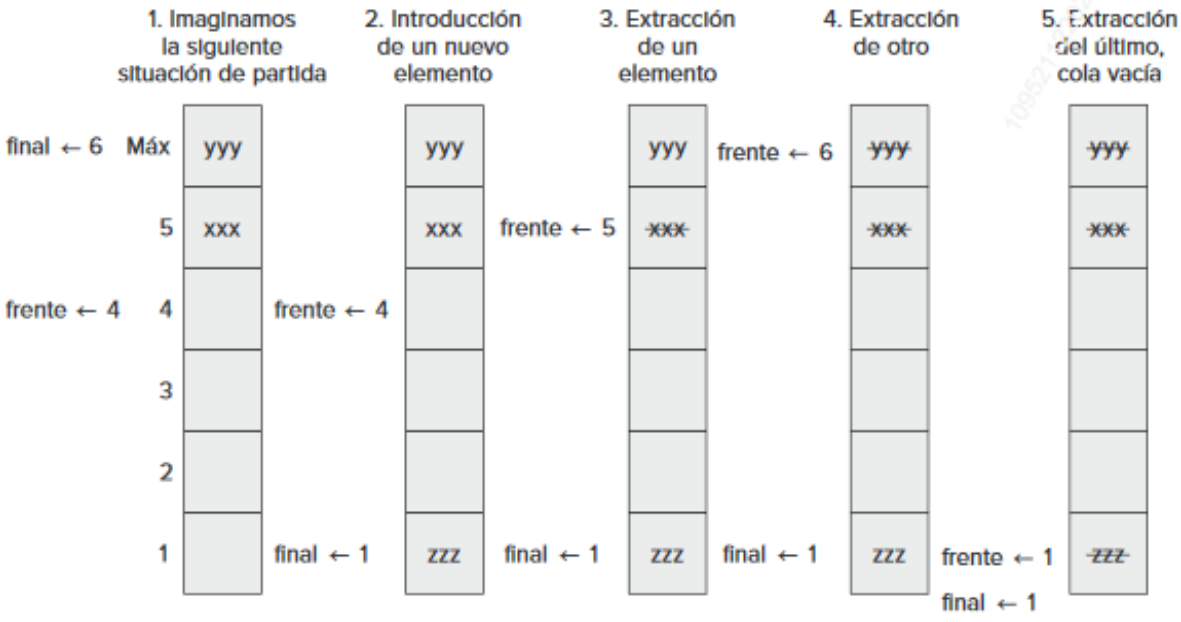
1. Retroceso.
   Consiste en mantener fijo a 1 el valor de `frente`, realizando un desplazamiento de una posición para todas las componentes ocupadas cada vez que se efectúa una supresión.
2. Reestructuración.
   Cuando `final` llega al máximo de elementos se desplazan las componentes ocupadas hacia atrás las posiciones necesarias para que el principio coincida con el principio de la tabla.
3. Mediante un array circular.

   Un array circular es aquel en el que se considera que la componente primera sigue a la componente última. Esta implementación obliga a dejar siempre una posición libre para separar el principio y el final del array. Evidentemente, seguirá existiendo la limitación de que pueda llenarse completamente el array, `Máx-1` posiciones ocupadas.

Cristian Pilapanta 1095211-2202112335

12.8.2. Aprovechamiento de la memoria

El mejor método para evitar el desaprovechamiento de espacio es el diseño de la cola mediante un array circular.



Deberemos efectuar las siguientes declaraciones:

```
const Máx = <expresión>
tipo
   registro : tipo_elemento
      ... : ...
      ... : ...
   fin_registro
   array[1..Máx] de tipo_elemento : arr
```

```
var
   entero            : frente, final
   arr               : c
   tipo_elemento     : elemento
```

En c[frente] estará siempre libre, sirviendo para separar el principio y el final del array.

```
procedimiento inicializar(S entero: frente,final)
inicio
   frente ← 1
   final ← 1
fin_procedimiento
```

Los elementos se añaden por el final.

```
procedimiento meter(E/S entero:final;  E/S arr: c;E        tipo_elemento: elemento)
inicio
   final ← final mod máx + 1
   c[final] ← elemento
fin_procedimiento
```

Los elementos se eliminan por el frente. El elemento a eliminar se encuentra siempre en la posición del array siguiente a la especificada por frente.

```
procedimiento sacar(E/S entero: frente; E arr: c;
                    S tipo_elemento: elemento)
inicio
   elemento ← c[frente mod máx + 1]
   frente ← frente mod máx + 1
fin_procedimiento
lógico función vacía(E entero: frente,final)
inicio
   si frente = final entonces
       devolver(verdad)
   si_no
       devolver(falso)
   fin_si
fin_función
```

Cuando la posición siguiente a final sea frente no podremos añadir más información, pues habría que hacerlo en:

c[final mod máx + 1] es decir c[frente]

y la cola se encontrará llena

```
lógico función llena(E entero: frente,final)
inicio
   si frente = (final mod Máx + 1!)  entonces
       devolver(verdad)
   si_no
       devolver(falso)
   fin_si
fin_función
procedimiento consultarPrimero(E entero: frente;  E arr: c;  S tipo_elemento: elemento)
inicio
   elemento ← c[frente mod máx + 1]
fin_procedimiento
```

Con esta estrategia, array circular, se sacrifica un elemento del array para distinguir la cola llena de cola vacía.

Cristian Pilaparta 1095211-2202112335

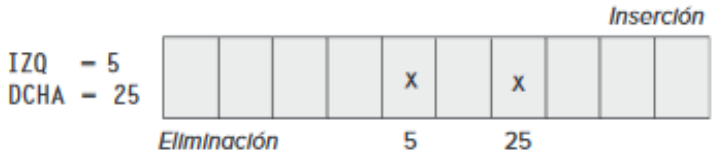
12.9. Doble cola

Existe una variante de la cola simple, estudiada anteriormente, y que es la *doble cola* o *bicola* que es una cola bidimensional en la que las inserciones y eliminaciones se pueden realizar en cualquiera de los dos extremos de la lista.

Figura 12.18. Doble cola (bicola).

Existen dos variantes de la doble cola:

- *Doble cola de entrada restringida*: acepta inserciones sólo al final de la cola.
- *Doble cola de salida restringida*: acepta eliminaciones sólo al frente de la cola.



Los procedimientos de inserción y eliminación de las dobles colas son variantes de los procedimientos estudiados para las colas simples y se dejan como ejercicio al lector.

Cristian Pilaparita 1095211-2202112335

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

12.1. *Una tienda de artículos deportivos desea almacenar en una lista enlazada, con un único elemento por producto, la siguiente información sobre las ventas realizadas: Código del artículo, Cantidad y Precio. Usando estructuras de tipo array, desarrollar un algoritmo que permita tanto la creación de la lista como su actualización al realizarse nuevas ventas o devoluciones de un determinado producto.*

Análisis del problema

El algoritmo contemplará la creación de la lista y colocará los elementos clasificados por código para que las búsquedas puedan resultar algo más rápidas. Al producirse una venta se han de considerar las siguientes posibilidades:

- Es la primera vez que se vende ese artículo y esto nos lleva a la inserción de un nuevo elemento en la lista.
- Ya se ha vendido alguna otra vez dicho artículo; por tanto, es una modificación de un elemento de la lista incrementándose la cantidad vendida.

Una devolución nos hará pensar en las siguientes situaciones:

- El comprador devuelve parte de lo que se había vendido de un determinado artículo, lo que representa una modificación de la cantidad vendida, decrementándose con la devolución.

- Se devuelve todo lo que se ha vendido de un determinado artículo y, en consecuencia, el producto debe desaparecer de la lista de ventas.

Diseño del algoritmo

```
algoritmo ejercicio_12_1
const
   máx = ...
tipo
   registro : tipo_elemento
    cadena: cod
    entero: cantidad
    real:   precio
   fin_registro
   registro : tipo_nodo
    tipo_elemento: elemento
    entero: sig
   fin_registro
  array[1..Máx] de tipo_nodo: lista
var
   entero: inic,vacío
   lista: m
   carácter: opción

inicio
   iniciar(m, vacío)
   inicializar(inic)
   repetir
    escribir('1.- Ventas')
    escribir('2.- Devoluciones')
    escribir('3.- Mostrar lista')
    escribir('4.- Fin')
    escribir('Elija opción')
    leer(opción)
    según_sea opción hacer
     '1':nuevasventas(inic,vacío,m)
     '2':devoluciones(inic,vacío,m)
     '3':recorrer(inic,m)
    fin_según
  hasta_que opción = '4'
fin

procedimiento inicializar(S entero: inic);
inicio
   inic ← 0
fin_procedimiento

lógico función vacía(E entero: inic)
inicio
   devolver(inic = 0)
fin_función
```

Cristian Pitpanta 1095211-2202112335

```
procedimiento iniciar(E/S lista: m; E/S entero: vacío)
var
   entero: i
inicio
   vacío ← 1
   desde i ← 1 hasta Máx-1 hacer
      m[i].sig ← i+1
   fin_desde
   m[Máx].sig ← 0
fin_procedimiento

procedimiento reservar(S entero: auxi; E lista: m; E/S entero: vacío)
inicio
   si vacío = 0 entonces
       escribir('Memoria agotada')
       auxi  ←  0
   si_no
       auxi  ←  vacio
       vacío  ←  m[vacío].sig
   fin_si
fin_procedimiento

lógico función llena(E entero: vacío)
inicio
   devolver(vacío = 0)
fin_función

procedimiento consultar(E entero: inic; S entero: posic, anterior;
                        E tipo_elemento: elemento; S lógico: encontrado;
                        E lista: m)
inicio
   si no vacía (inic) entonces
      anterior  ←  0
      posic  ←  inic
      mientras (m[posic].elemento.cod < elemento.cod) y (posic<>0) hacer
        anterior  ←  posic
        posic  ←  m[posic].sig
      fin_mientras
      si m[posic].elemento.cod = elemento.cod entonces
          encontrado  ←  verdad
      si_no
          encontrado  ←  falso
      fin_si
fin_si
fin_procedimiento

procedimiento insertar(E/S entero: inic,anterior;
                       E tipo_elemento: elemento;
                       E/S lista: m; E/S entero: vacío)
var
   entero: auxi
inicio
   si no llena (vacío) entonces
     reservar(auxi,m,vacío)
     m[auxi].elemento  ←  elemento
```

Cristian Pilaparia 1095211-2202112335

```
      si anterior=0 entonces
          m[auxi].sig  ←  inic
          inic  ←  auxi
      si_no
          m[auxi].sig  ←  m[anterior].sig
          m[anterior].sig  ←  auxi
      fin_si
      anterior  ←  auxi
fin_si
fin_procedimiento

procedimiento escribir_reg(E tipo_elemento: e)
inicio
   escribir(e.cod)
   escribir(e.cantidad)
   escribir(e.precio)
fin_procedimiento

procedimiento recorrer(E entero: inic; E lista: m)
var
   entero: posic
inicio
   posic ← inic
   mientras posic<>0 hacer
      escribir_reg(m[posic].elemento)
      posic ← m[posic].sig
   fin_mientras
fin_procedimiento

procedimiento liberar(E/S entero: posic; E/S lista: m;
                      E/S entero: vacío)
inicio
   m[posic].sig ← vacío
   vacío ← posic
fin_procedimiento

procedimiento suprimir(E/S entero: inic, anterior, posic;
                       E/S lista: m; E/S entero: vacío)
inicio
   si anterior=0 entonces
       inic ← m[posic].sig
   si_no
       m[anterior].sig ← m[posic].sig
   fin_si
   liberar(posic,m,vacío)
   anterior ← 0
   posic ← inic
fin_procedimiento

procedimiento nuevasventas(E/S entero: inic, vacío; E/S lista: m)
var
   tipo_elemento: elemento
   lógico: encontrado
   entero: anterior, posic
```

Cristian Pitapanta 1095211-2202112335

```
inicio
   repetir
     escribir('Introduzca * en el código para terminar')
     escribir('Código:   ')
     leer(elemento.cod)
     si elemento.cod <>'*' entonces
       si vacía(inic) entonces
           anterior ← 0
           escribir('Cantidad: ')
           leer(elemento.cantidad)
           escribir('Precio:   ')
           leer(elemento.precio)
           insertar(inic,anterior,elemento,m,vacío)
       si_no
           consultar(inic,posic,anterior,elemento,encontrado,m)
           si no encontrado entonces
              si no llena(vacío) entonces
                  escribir('Cantidad: ')
                  leer(elemento.cantidad)
                  escribir('Precio:   ')
                  leer(elemento.precio)
                  insertar(inic,anterior,elemento,m,vacío)
              si_no
                  escribir('Llena')
              fin_si
           si_no
               escribir('Cantidad:')
               leer(elemento.cantidad)
               m[posic].elemento.cantidad ← m[posic].elemento.cantidad+
                                            elemento.cantidad
           fin_si
       fin_si
     fin_si
   hasta_que elemento.cod = '*'
fin_procedimiento

procedimiento devoluciones(E/S entero: inic, vacío; E/S lista: m)
var
   tipo_elemento: elemento
   entero       : posic, anterior
   entero       : cantidad
   lógico       : encontrado
inicio
   si no vacía(inic) entonces
     escribir('Introduzca un * en el código para terminar')
     escribir('Código: ')
     leer(elemento.cod)
   si_no
     escribir('No hay ventas, no puede haber devoluciones')
   fin_si
```

Cristian Pilapanta 1095211-2202112335

```
  mientras (elemento.cod<>'*' ) y no vacía(inic) hacer
    consultar(inic, posic, anterior, elemento, encontrado, m)
    si encontrado entonces
      repetir
        escribir('Deme cantidad devuelta ')
        leer(cantidad)
        si cantidad > m[posic].elemento.cantidad entonces
          escribir('Error')
        fin_si
      hasta_que cantidad <= m[posic].elemento.cantidad
      m[posic].elemento.cantidad ← m[posic].elemento.cantidad-cantidad
      si m[posic].elemento.cantidad=0 entonces
        suprimir(inic,anterior,posic,m,vacío)
      fin_si
    si_no
      escribir('No existe')
    fin_si
    si no vacía(inic) entonces
      escribir('Introduzca un * en el código para terminar')
      leer(elemento.cod)
    si_no
      escribir('No hay ventas, no puede haber devoluciones')
    fin_si
  fin_mientras
fin_procedimiento
```

12.2. *Diseñar un procedimiento que realice una copia de una pila en otra.*

Análisis del problema

Entendemos por copiar la acción de rellenar otra pila con los mismos elementos y en el mismo orden. Por lo tanto, si simplemente sacamos los elementos de la pila y los metemos en otra, tendrá los mismos elementos, pero en orden distinto. Tenemos dos soluciones, una sería utilizar una pila auxiliar: sacaremos los elementos de la pila principal y los meteremos en la auxiliar, para después volcarlos en dos pilas, una de las cuales es la de salida. La otra solución sería recursiva: se sacan los elementos de la pila mediante llamadas recursivas; cuando la pila esté vacía inicializaremos la copia y a la vuelta de la recursividad se van introduciendo los elementos en dos pilas en orden inverso a como han salido.

Observe que el procedimiento valdrá tanto para la implementación con arrays como con estructuras dinámicas de datos.

Diseño del algoritmo

Solución iterativa

```
procedimiento CopiarPila(E/S pila: p; S pila: copia)
  var
    pila: aux
    tipo_elemento: e
  inicio
   PilaVacía(aux)
   mientras no EsPilaVacía(p) hacer
     Tope(p,e)          "extra el primer elemento sin borrado
     PInsertar(aux,e)   "inserta elemento
     PBorrar(p)         "borra elemento
   fin_mientras
   PilaVacía(copia)
```

Cristian Pilaparita 1095211-2202112335

```
    mientras no EsPilaVacía(aux) hacer
      Tope(aux,e)
      PInsertar(copia, e)
      PInsertar(p, e)
      PBorrar(aux)
    fin_mientras
  fin_procedimiento
```

Solución recursiva (la recursividad se estudia en el capítulo 14)

```
  procedimiento CopiarPilaR(E/S pila: p, copia)
    var
      tipo_elemento: e
    inicio
      si no EsPilaVacía(p) entonces
        Tope(p,e)
        PBorrar(p)
        CopiarPila(p, copia)
        PInsertar(copia, e)
        Pinsertar(p, e)
      si_no
        PilaVacía(copia)
      fin_si
    fin_procedimiento
```

Los procedimientos y funciones utilizados implementados con punteros son:

```
procedimiento PilaVacía(S pila: p)
  inicio
    p ← nulo
  fin_procedimiento

lógico función EsPilaVacía(E pila: p)
  inicio
     devolver(p = nulo)
  fin_función

procedimiento PInsertar(E/S pila: p; E tipo_elemento: e)
  var
    pila: aux
  inicio
    reservar(aux)
    aux→.info ← e
    aux→.cimaant ← p
    p ← aux
  fin_procedimiento

procedimiento PBorrar(E/S pila: p)
  var
    pila: aborrar
  inicio
    Si no EsPilaVacía (p) entonces
       aborrar ← p
       p ← p→.cimaant
       liberar(aborrar)
    fin_sin
  fin_procedimiento
```

Cristian Pilaparia 1095211-2202112335

```
procedimiento Tope(E pila: p; S tipo_elemento: e)
   inicio
      e ← p→.info
   fin_procedimiento
```

12.3. *Diseñar un procedimiento que elimine el elemento enésimo de una pila.*

Análisis del problema

También en este caso se debe utilizar una pila auxiliar o recursividad para poder restaurar los elementos en el mismo orden. Es necesario borrar elementos e insertarlos en la pila auxiliar hasta llegar al elemento n. En ese punto, se sacan todos los elementos de la pila auxiliar y se introducen en la pila original. Obsérvese que también en este caso es totalmente indistinto utilizar estructuras de datos dinámicas o estáticas.

Diseño del algoritmo

Solución iterativa

```
procedimiento BorrarElementoN(E/S pila: p; E entero: n)
   var
     pila: aux
     tipo_elemento: e
     entero: i
   inicio
     i ← 1
   PilaVacía(aux)
   mientras no EsPilaVacía(p) y (i<n) hacer
     i ← i+1
     Tope(p, e)
     PInsertar(aux, e)
     PBorrar(p)
   fin_mientras
   Pborrar(p)
   mientras no EsPilaVacía(aux) hacer
     Tope(aux, e)
     PInsertar(p, e)
     PBorrar(aux)
   fin_mientras
fin_procedimiento
```

Solución recursiva

```
procedimiento BorrarElementoN(E/S pila: p; E entero: n)
   var
      tipo_elemento: e
   inicio
      si (n>1) y no EsPilaVacía(p) entonces
        tope(p,e)
        PBorrar(p)
        BorrarElementoN(p, n-1)
        PInsertar(p,e)
      si_no
        PBorrar(p)
      fin_si
   fin_procedimiento
```

Cristian Pilapaña 1095211-2202112335

12.4. *Diseñar un algoritmo para comprobar, utilizando pilas y colas, si una frase es un palíndromo (un palíndromo es una frase que se lee igual de izquierda a derecha que de derecha a izquierda).*

Análisis del problema

Se puede aprovechar el distinto orden en que salen los elementos de una pila y una cola para averiguar si una frase es igual a ella misma invertida. Para ello, una vez introducida la frase, se insertan en una pila y una cola todos los caracteres (se evitarán los signos de puntuación, y se podría mejorar si se convierten todos los caracteres a mayúsculas o minúsculas y se eliminan los acentos).

A continuación se van sacando elementos de la pila y de la cola. Si aparece algún carácter distinto, es que la frase no es igual a ella misma invertida y, por lo tanto, no será un palíndromo. Si al acabar de sacar los elementos, todos han sido iguales, se trata de un palíndromo.

Diseño del algoritmo

```
algoritmo ejercicio_12_4;
   {Aquí deberían incluirse las declaraciones, procedimientos y funciones
   para trabajar con pilas y colas.
   Es indistinto trabajar con estructuras estáticas o dinámicas}
   var
      pila    : p
      cola    : c
      cadena  : car1, car2, frase
      entero  : i

   inicio
      ColaVacía(c)
      PilaVacía(p)
      leer(frase)
      desde i ← 1 hasta longitud(frase) hacer
         car1 ← subcadena(frase, i, 1)
         {si no es un signo de puntuación}
         si posicion(car1,',:;.')= 0 entonces
            CInsertar(c,car1)
            PInsertar(p,car1)
         fin_si
      fin_desde
      repetir
         Primero(c,car1)
         Tope(p,car2)
         PBorrar(p)
         CBorrar(c)
      hasta_que (car1<>car2) o EsColaVacía(c)
      si car1=car2 entonces
         escribir('Es un palíndromo')
      si_no
         escribir('No es un palíndromo')
      fin_si
fin
```

Cristian Pilaparita 1095211-2202112335

CONCEPTOS CLAVE

- Apuntador
- Cola
- Doble cola
- Enlace
- Estructura de datos dinámica
- Estructura de datos estática
- Lista circular
- Lista doblemente enlazada
- Lista enlazada
- Pila
- Puntero

RESUMEN

Una **lista lineal** es una lista en la que cada elemento tiene un único sucesor. Las operaciones típicas en una lista lineal son: inserción, supresión, recuperación y recorrido.

Una **lista enlazada** es una colección ordenada de datos en los que cada elemento contiene la posición (dirección) del siguiente elemento. Es decir, cada elemento (nodo) de la lista contiene dos partes: datos y enlace (puntero).

Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo. Cuando se desea insertar un elemento en una lista enlazada, se deben considerar dos casos: añadir al principio y añadir en el interior o añadir al final. Si se desea eliminar un nodo de una lista se deben considerar dos casos: eliminar el primer nodo y eliminar cualquier otro nodo. El recorrido de una lista enlazada implica visitar cada nodo de la lista y procesar en su caso.

Una **lista doblemente enlazada** es una lista en la que cada nodo tiene un puntero a su sucesor y otro a su predecesor. Una **lista enlazada circularmente** es una lista en la que el enlace del último nodo apunta al primero de la lista.

Una **pila** es una estructura de datos tipo **LIFO** (***last-in, first-out,*** último en entrar, primero en salir) en la que los datos se insertan y eliminan por el mismo extremo que se denomina *cima de la pila.* Se definen diferentes operaciones: `crear`, `apilar`, `desapilar`, `pilaVacía`, `pilaLlena`, `cimaPila`.

Una **cola** (**FIFO,** first-in, first-out) es una lista lineal en la que los datos se pueden insertar por un extremo denominado *Cabeza* y se elimina o borra por el otro extremo denominado *Cola* o *Final.* Las operaciones básicas de una `cola` son: `poner`, `quitar`, `frenteCola` y `Colavacía`, `Colallena`.

Las pilas y las colas se pueden implementar mediante arrays y mediante listas enlazadas.

EJERCICIOS

12.1. Dada una lista lineal cuya estructura de nodos consta de los campos `INFO` y `ENLACE`, diseñar un algoritmo que cuente el número de nodos de la lista.

12.2. Diseñar un algoritmo que cambie el campo `INFO` del *n-ésimo* nodo de una lista enlazada simple por un valor dado *x*.

12.3. Dadas dos listas enlazadas, cuyos nodos frontales se indican por los apuntadores `PRIMERO` y `SEGUNDO`, respectivamente, realizar un algoritmo que una ambas listas. El nodo frontal de la lista nueva se almacenará en `TERCERO`.

12.4. Se dispone de una lista enlazada `DEMO 1` almacenada en memoria. Realizar un algoritmo que copie la lista `DEMO 1` en otra denominada `DEMO 2`.

12.5. Escribir un algoritmo que realice una inserción contigua a la izquierda del *n-ésimo* nodo de una lista enlazada y repetir el ejercicio para una inserción también contigua a la derecha de *n-ésimo* nodo.

12.6. Escribir un algoritmo que divida una lista enlazada determinada en dos listas enlazadas independientes.

Cristian Pilapanta 1095211-2202112335

El primer nodo de la lista principal es PRIMERO y la variable PARTIR es la dirección del nodo que se convierte en el primero de los nodos de la segunda lista enlazada resultante.

12.7. Como aplicación de pilas, obtener un subalgoritmo función recursiva de la función de Ackermann.

Función de Ackermann:

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1,1) & \text{si } n = 0 \\ A(m - 1)\ A(m, n - 1) & \text{restantes casos} \end{cases}$$

12.8. Escribir un subalgoritmo que permita insertar un elemento en una doble cola —representada por un vector—. Tengan en cuenta que debe existir un parámetro que indique el extremo de la doble cola en que debe realizarse la inserción.

12.9. Realizar un algoritmo que cuente el número de nodos de una lista circular que tiene una cabecera.

12.10. Diseñar un algoritmo que inserte un nodo al final de una lista circular.

Cristian Pilaparta 1095211-2202112335

CAPÍTULO 13

Estructura de datos no lineales (árboles y grafos)

13.1. Introducción
13.2. Árboles
13.3. Árbol binario
13.4. Árbol binario de búsqueda
13.5. Grafos

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Cristian Pilapaña 1095211-2202112335

1095211220211235

INTRODUCCIÓN

Las *estructuras dinámicas lineales de datos —listas enlazadas, pilas y colas—* tienen grandes ventajas de flexibilidad sobre las representaciones contiguas; sin embargo, tienen un punto débil: son listas secuenciales, es decir, están dispuestas de modo que es necesario moverse a través de ellas una posición cada vez (cada elemento tiene un siguiente elemento). Esta linealidad es típica de cadenas, de elementos que pertenecen a una sola dimensión: campos en un registro, entradas en una pila, entradas en una cola y de nodos en una lista enlazada simple. En este capítulo se tratarán las *estructuras de datos no lineales* que resuelven los problemas que plantean las listas lineales y en las que cada elemento puede tener diferentes "siguientes" elementos, que introducen el concepto de estructuras de bifurcación. Estos tipos de datos se llaman *árboles*.

Asimismo, este capítulo introduce a una estructura matemática importante que tiene aplicaciones en ciencias tan diversas como la sociología, química, física, geografía y electrónica. Estas estructuras se denominan *grafos*.

13.1. Introducción

Las estructuras de datos que han sido examinadas hasta ahora en este libro son lineales. A cada elemento le correspondía siempre un "siguiente" elemento. La linealidad es típica de cadenas, de elementos de arrays o listas, de campos en registros, entradas en pilas o colas y nodos en listas enlazadas.

En este capítulo se examinarán las estructuras de datos *no lineales*. En estas estructuras cada elemento puede tener diferentes "siguientes" elementos, lo que introduce nos lleva al concepto de estructuras de bifurcación.

Las estructuras de datos no lineales son *árboles* y *grafos*. A estas estructuras se les denomina también *estructuras multienlazadas.*

13.2. Árboles

El árbol es una estructura de datos fundamental en informática, (ingeniería de sistemas) muy utilizada en todos sus campos, porque se adapta a la representación natural de informaciones homogéneas organizadas y de una gran comodidad y rapidez de manipulación. Esta estructura se encuentra en todos los dominios (campos) de la informática, desde la pura *algoritmia* (métodos de clasificación y búsqueda) a la *compilación* (árboles sintácticos para representar las expresiones o producciones posibles de un lenguaje) o incluso los dominios de la inteligencia artificial (árboles de juegos, árboles de decisiones, de resolución, etcétera).

Las estructuras tipo árbol se usan principalmente para representar datos con una relación jerárquica entre sus elementos, como son árboles genealógicos, tablas, entre otros.

Un árbol A es un conjunto finito de uno o más nodos, tales que:

1. Existe un nodo especial denominado RAÍZ(v_1) del árbol.
2. Los nodos restantes (v_2, v_3, ..., v_n) se dividen en $m >= 0$ conjuntos disjuntos denominados A_1, A_2, ..., A_m, cada uno de los cuales es, a su vez, un árbol. Estos árboles se llaman *subárboles* del RAÍZ.

La definición de árbol implica una estructura recursiva; esto es, la definición del árbol se refiere a otros árboles.

Un árbol con ningún nodo es un *árbol nulo*, no tiene raíz.

La figura 13.1 muestra un árbol en el que se ha rotulado cada nodo con una letra dentro de un círculo; ésta es una notación típica para dibujar árboles.

Los tres subárboles del raíz A son B, C y D, respectivamente. B es la raíz de un árbol con un subárbol E. Este subárbol no tiene subárbol conectado. El árbol C tiene dos subárboles, F y G.

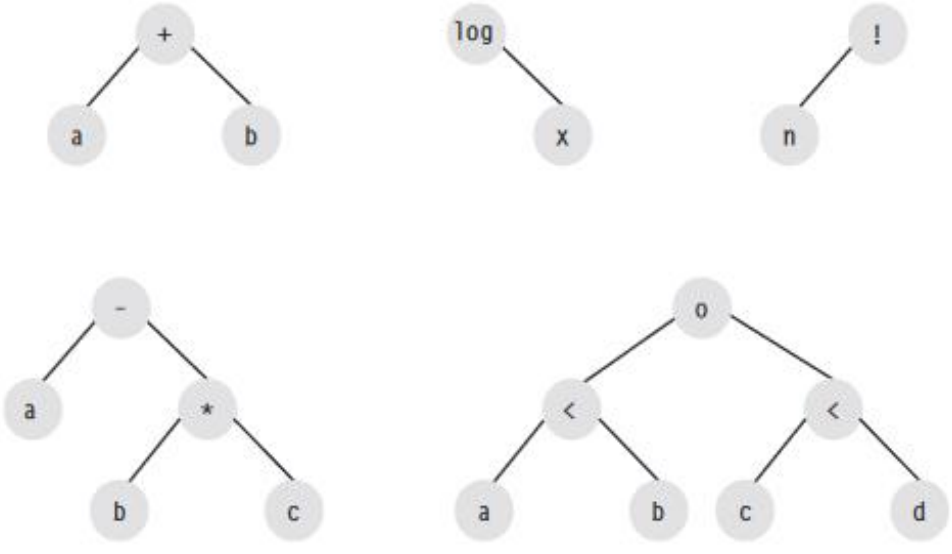


Figura 13.1. Diferentes árboles.

Cristian Pilapanta 1096211-2202112335

13.2.1. Terminología y representación de un árbol general

La representación y terminología de los árboles se realiza con las típicas notaciones de las relaciones familiares en los árboles genealógicos: padre, hijo, hermano, ascendente, descendiente, etcétera; observe el árbol general de la figura 13.2.

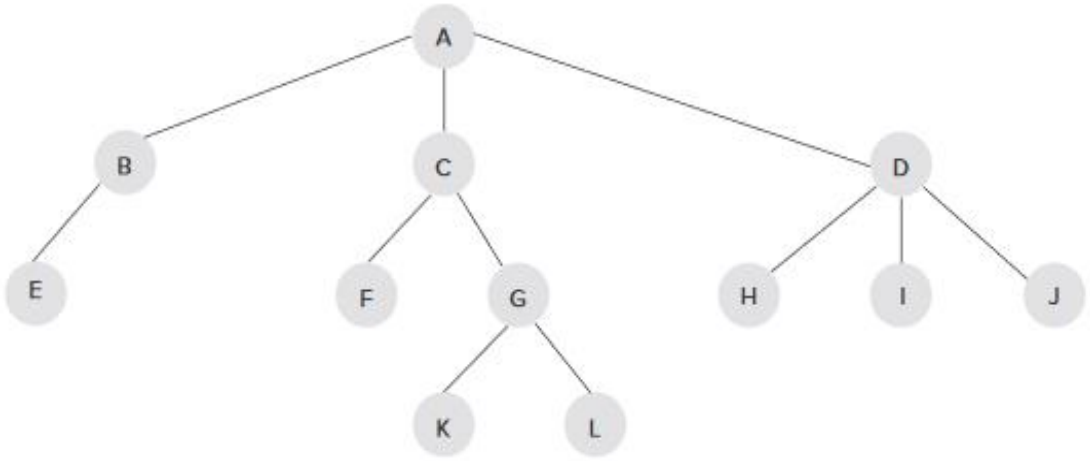


Figura 13.2. Árbol general.

Cristian Pilapanta 1095211-2202112335

Las definiciones a tener en cuenta son:

- *Raíz* del árbol. Todos los árboles que no están vacíos tienen un único nodo raíz. Todos los demás elementos o nodos se derivan o descienden de él. El nodo raíz no tiene *padre*, es decir, no es el hijo de ningún elemento.
- *Nodo*, son los vértices o elementos del árbol.
- *Nodo terminal* u *hoja (leaf node)* es aquel nodo que no contiene ningún subárbol (los nodos terminales u hojas del árbol de la figura 13.2 son E, F, K, L, H y J).
- A cada nodo que no es hoja se asocia uno o varios subárboles llamados *descendientes (offspring)* o *hijos*. De igual forma, cada nodo tiene asociado un antecesor o *ascendiente* llamado *padre*.
- Los nodos de un mismo padre se llaman *hermanos*.
- Los nodos con uno o dos subárboles —no son hojas ni raíz— se llaman *nodos interiores* o *internos*.
- Una colección de dos o más árboles se llama *bosque (forest)*.
- Todos los nodos tienen un solo padre —excepto el *raíz*— que no tiene padre.
- Se denomina *camino* el enlace entre dos nodos consecutivos y *rama* es un camino que termina en una hoja.
- Cada nodo tiene asociado un número de *nivel* que se determina por la longitud del camino desde el raíz al nodo específico. Por ejemplo, en el árbol de la figura 13.2.

Nivel 0	A
Nivel 1	B, C, D
Nivel 2	E, F, G, H, I, J
Nivel 3	K, L

- La *altura* o *profundidad* de un árbol es el número máximo de nodos de una rama, que equivale al nivel más alto de los nodos más uno. El *peso* de un árbol es el número de nodos terminales, la *altura* y el *peso* del árbol de la figura 13.2 son 4 y 7, respectivamente.

Las representaciones gráficas de los árboles —además de las ya expuestas— pueden ser las mostradas en la figura 13.3.

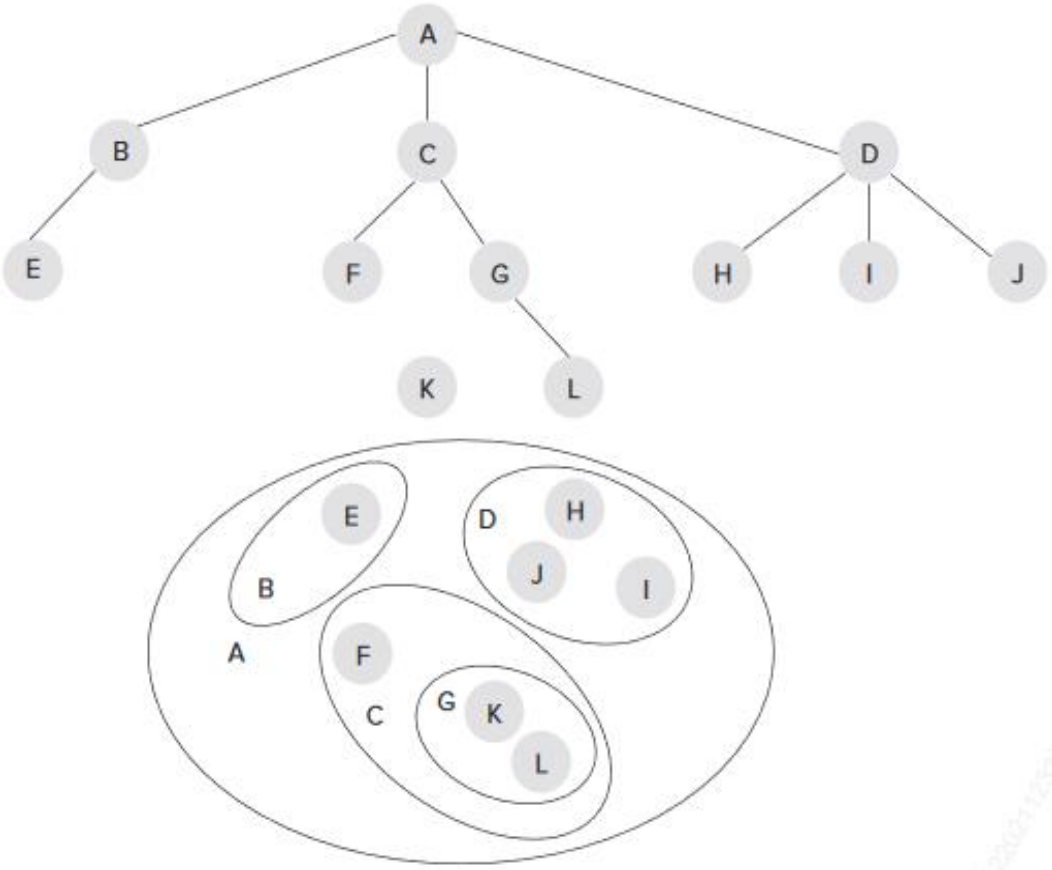


Figura 13.3. Representaciones de árboles.

Cristian Pilaparia 1095211-2202112336

13.3. Árbol binario

Existe un tipo de árbol denominado *árbol binario* que puede ser implementado fácilmente en una computadora.

Un *árbol binario* es un conjunto finito de cero o más nodos, tales que:

- Existe un nodo denominado raíz del árbol.
- Cada nodo puede tener 0, 1 o 2 subárboles, conocidos como *subárbol izquierdo* y *subárbol derecho*. La figura 13.4 representa diferentes tipos de árboles binarios:

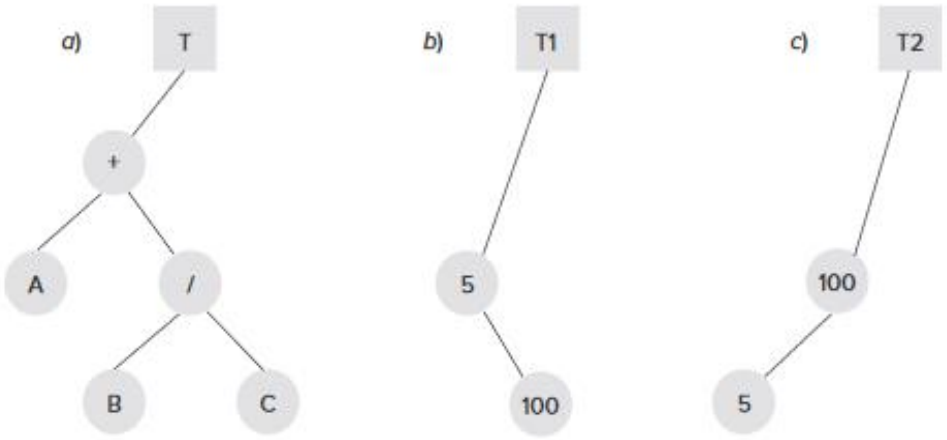


Figura 13.4. Ejemplos de árboles binarios: *a*) expresión árbol a + b/c; *b*) y *c*) dos árboles diferentes con valores enteros.

13.3.1. Terminología de los árboles binarios

Dos árboles binarios se dice que son *similares* si tienen la misma estructura, y son *equivalentes* si son similares y contienen la misma información (figura 13.5).

Un árbol binario está *equilibrado* si las alturas de los dos subárboles de cada nodo se diferencian en una unidad como máximo.

altura (subárbol izquierdo) – altura (subárbol derecho) ≤ 1

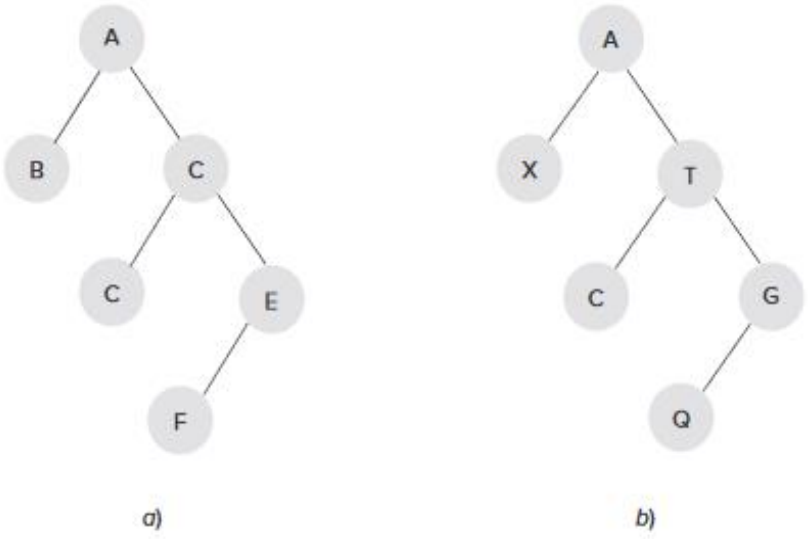


Figura 13.5. Árboles binarios: *a*) similares, *b*) equivalentes.

El procesamiento de árboles binarios equilibrados es más sencillo que los árboles no equilibrados. En la figura 13.6 se muestran dos árboles binarios de diferentes alturas y en la figura 13.7, árboles equilibrados y sin equilibrar.

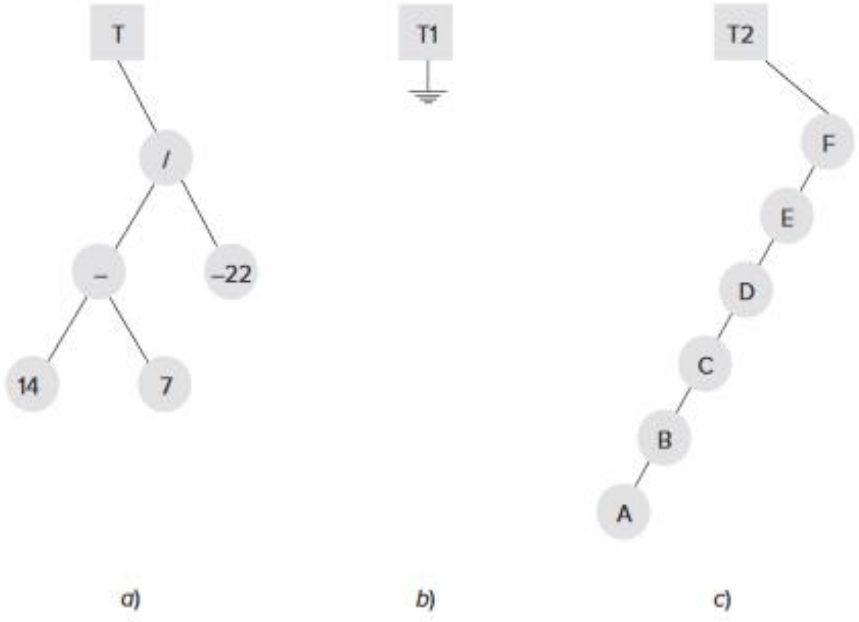


Figura 13.6. Árboles binarios de diferentes alturas: *a*) altura 3, *b*) árbol vacío, altura 0, *c*) altura 6.

Cristian Pilaparta 1095211-2202112336

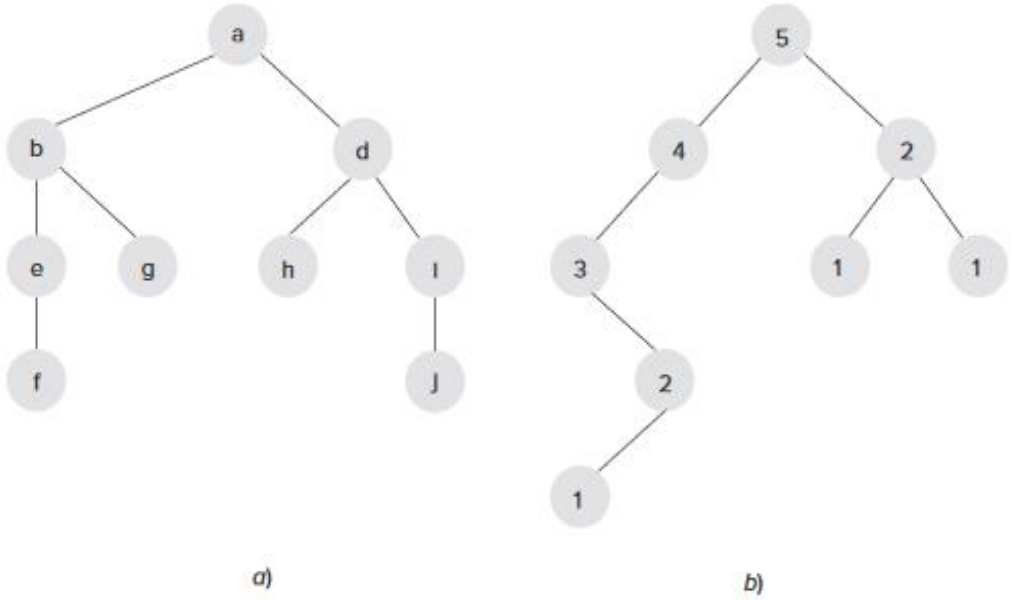


Figura 13.7. Árboles binarios: *a*) equilibrados, *b*) no equilibrados.

Cristian Pilapaña 1095211-2202112336

13.3.2. Árboles binarios completos

Un árbol binario se llama *completo* si todos sus nodos tienen exactamente dos subárboles, excepto los nodos de los niveles más bajos que tienen cero. Un árbol binario completo, tal que todos los niveles están llenos, se llama *árbol binario lleno*.

En la figura 13.8 se ilustran ambos tipos de árboles.

Un árbol binario T de nivel h puede tener como máximo $2^h - 1$ nodos.

La altura de un árbol binario lleno de n nodos es $\log_2(n + 1)$. A la inversa, el *número máximo de nodos* de un árbol binario de altura h será $2^h - 1$. En la figura 13.9 se muestra la relación matemática que liga los nodos de un árbol.

Por último, se denomina *árbol degenerado* un árbol en el que todos sus nodos tienen solamente un subárbol, excepto el último.

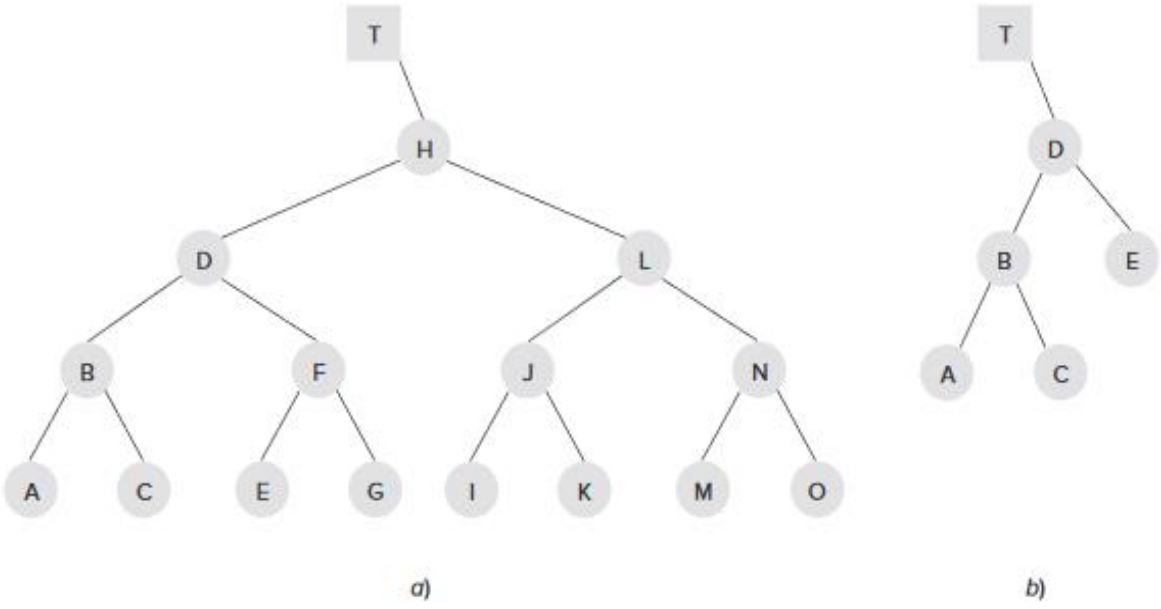


Figura 13.8. *a*) árbol binario lleno de altura 4, *b*) árbol binario completo de altura 3.

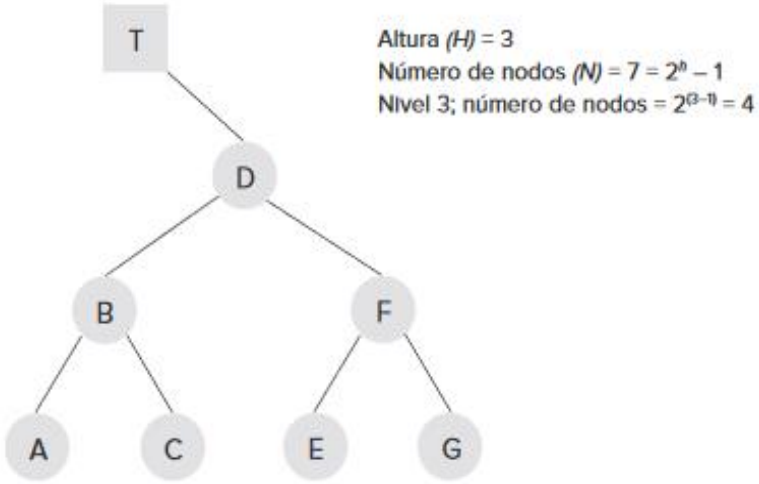


Figura 13.9. Relaciones matemáticas de un árbol binario.

13.3.3. Conversión de un árbol general en árbol binario

Dado que los árboles binarios son la estructura fundamental en la teoría de árboles, será preciso disponer de algún mecanismo que permita la conversión de un árbol general en un árbol binario.

Los árboles binarios son más fáciles de programar que los generales. En éstos es imprescindible deducir cuántas ramas o caminos se desprenden de un nodo en un momento dado. Por ello, y debido a que de los árboles binarios siempre se cuelgan como máximo dos subárboles, su programación será más sencilla.

Afortunadamente existe una técnica para convertir un árbol general a formato de árbol binario. Supongamos que tiene el árbol A y quiere convertirlo en un árbol binario B. El algoritmo de conversión tiene tres pasos fáciles:

1. La raíz de B es la raíz de A.
2. *a*) Enlazar al nodo raíz con el camino que conecta el nodo más a la izquierda (su hijo).
   b) Enlazar este nodo con los restantes descendientes del nodo raíz en un camino, con lo que se forma el nivel 1.
   c) A continuación, repetir los pasos *a*) y *b*) con los nodos del nivel 2, enlazando siempre en un mismo camino todos los hermanos —descendientes del mismo nodo—. Repetir estos pasos hasta llegar al nivel más alto.
3. Girar el diagrama resultante 45° para diferenciar entre los subárboles izquierdo y derecho.

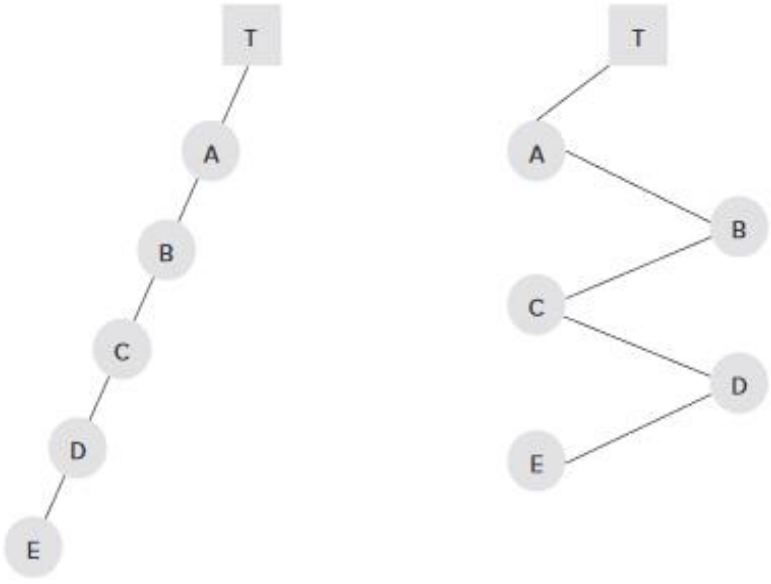
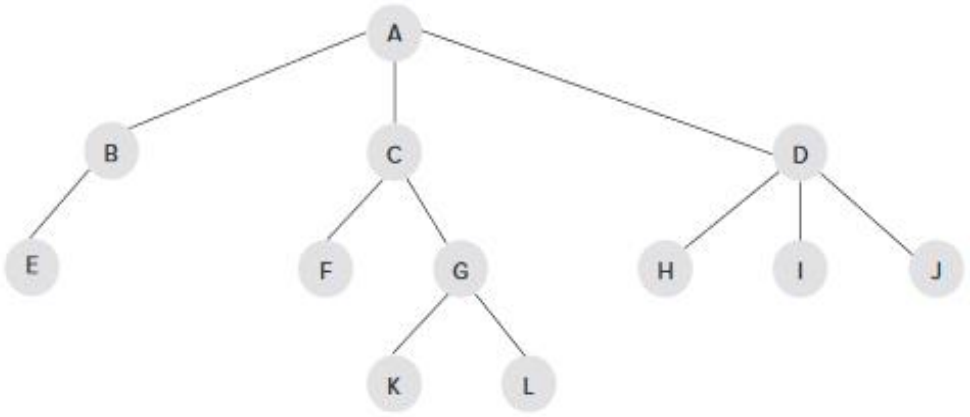


Figura 13.10. Árboles degenerados.

Cristian Pilapanta 1095211-2202112336

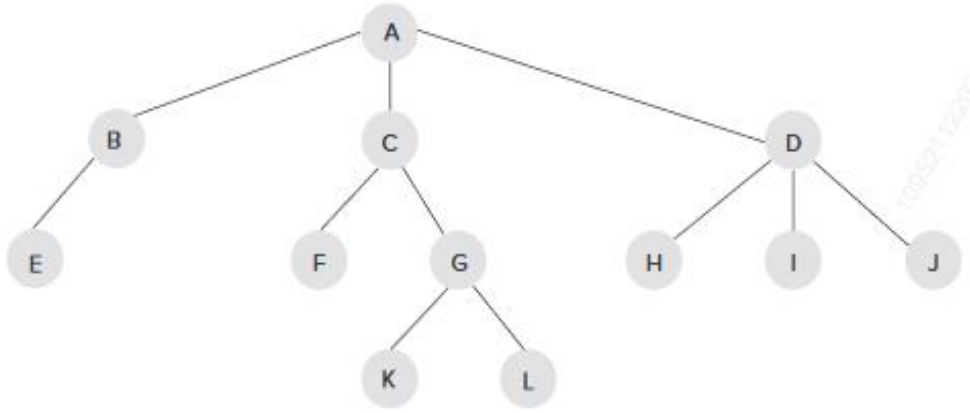
EJEMPLO 13.1

Convertir el árbol general T en un árbol binario.

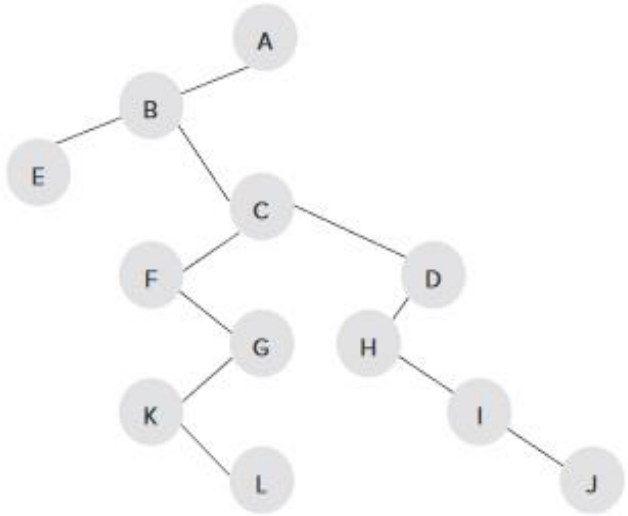


Siguiendo los pasos del algoritmo.

Paso 1



Paso 2



Cristian Pilapaña 1095211-2202112336

Paso 3

Obsérvese que no existe camino entre E y F, debido a que no son descendientes del árbol original, ya que ellos tienen diferentes padres B y C.

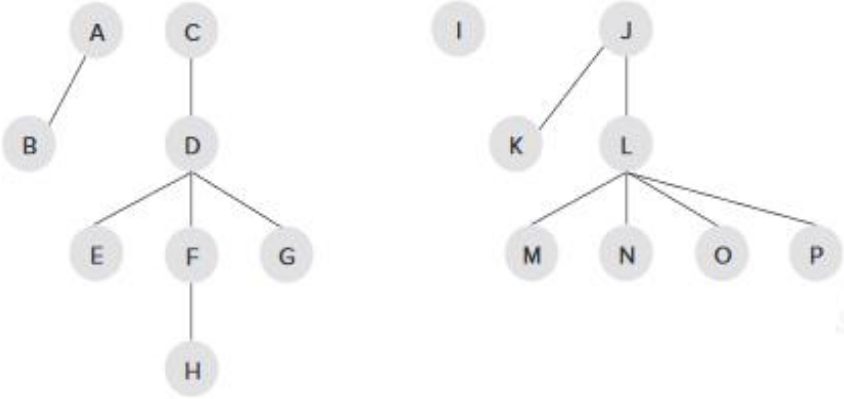
En el árbol binario resultante los punteros izquierdos son siempre de un nodo padre a su primer hijo (más a la izquierda) en el árbol general original. Los punteros derechos son siempre desde un nodo de sus descendientes en el árbol original.

EJEMPLO 13.2

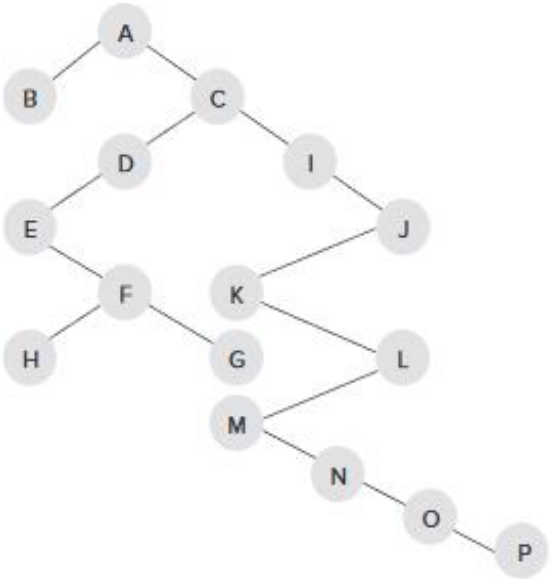
El algoritmo de conversión puede ser utilizado para convertir un bosque de árboles generales a un solo árbol binario.

El bosque siguiente puede ser representado por un árbol binario.

Bosque de árboles



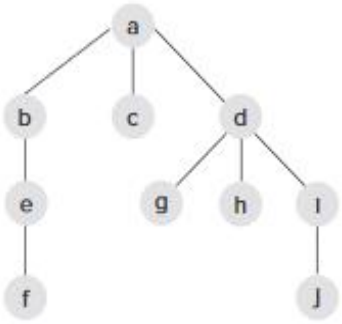
Árbol binario equivalente



Cristian Pilaparita 1095211-2202112336

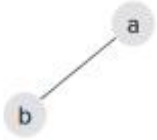
EJEMPLO 13.3

Convertir el árbol general en árbol binario.

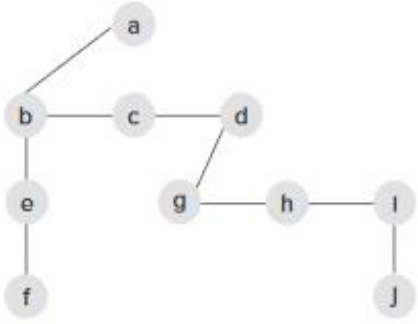


Solución

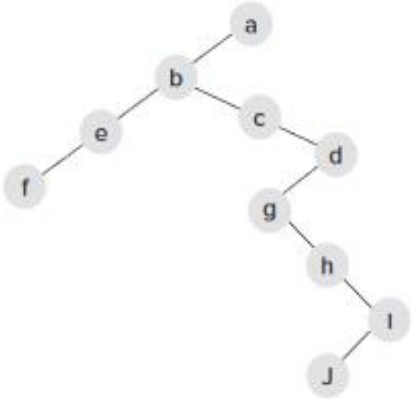
Paso 1



Paso 2



Paso 3



Cristian Pilaparda 1095211-2202112336

13.3.4. Representación de los árboles binarios

Los árboles binarios pueden ser representados de dos modos diferentes:

- *Mediante punteros* o *apuntadores* (lenguajes C y C++).
- *Mediante arrays* o *listas enlazadas.*
- *Vinculando nodos, objetos* con miembros que referencian otros objetos del mismo tipo.

13.3.4.1. Representación por punteros

Cada nodo de un árbol será un registro que contiene al menos tres campos:

- Un campo de datos con un tipo de datos.
- Un puntero al nodo del subárbol izquierdo (que puede ser **nulo**-`null`).
- Un puntero al nodo del subárbol derecho (que puede ser **nulo**-`null`).

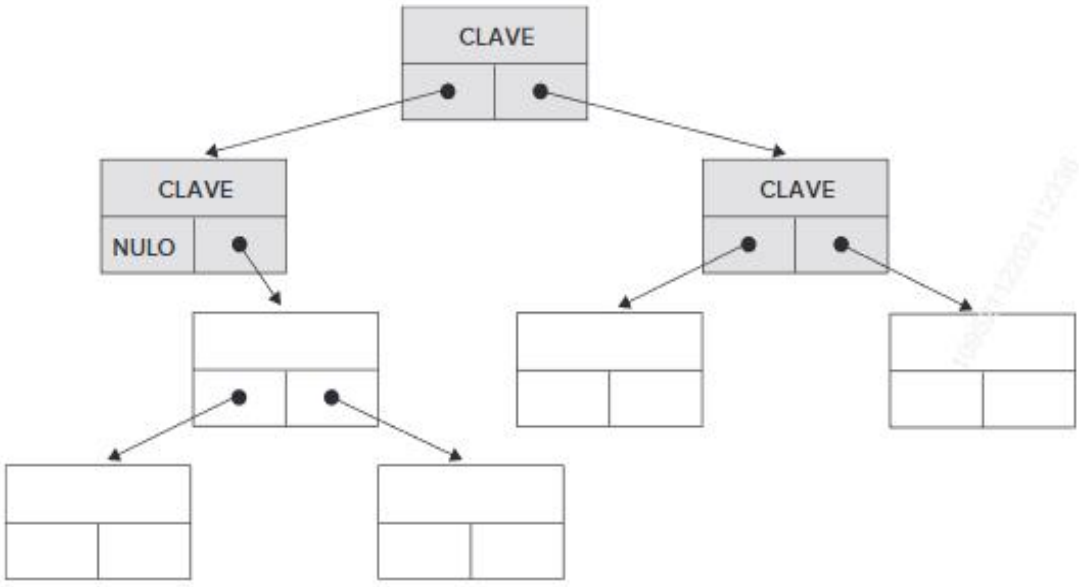
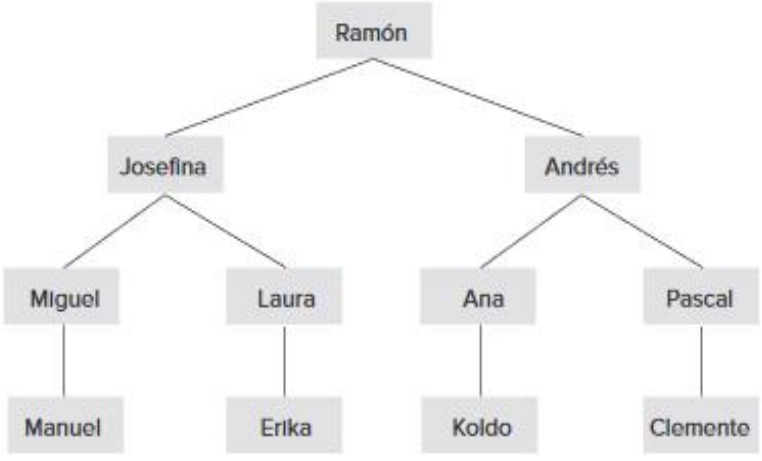


Figura 13.11. Representación de un árbol con punteros.



Cristian Pilapanta 1095211-2202112336

En lenguaje algorítmico se tendrá:

```
tipo nodo_árbol
   puntero_a nodo_árbol: punt
   registro : nodo_árbol
      <tipo_elemento> : elemento
      punt: subiz, subder
   fin_registro
```

13.3.4.2. Representación por listas enlazadas

Mediante una lista enlazada se puede siempre representar el árbol binario de la figura 13.12.

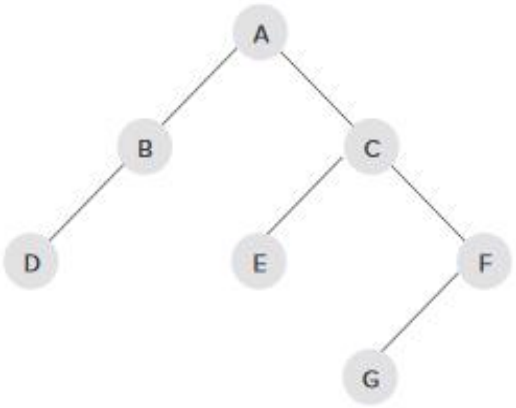


Figura 13.12. Árbol binario.

Nodo del árbol: campo 1 INFO (nodo)
campo 2 IZQ (nodo)
campo 3 DER (nodo)

El árbol binario representado como una lista enlazada se puede ver en la figura 13.13.

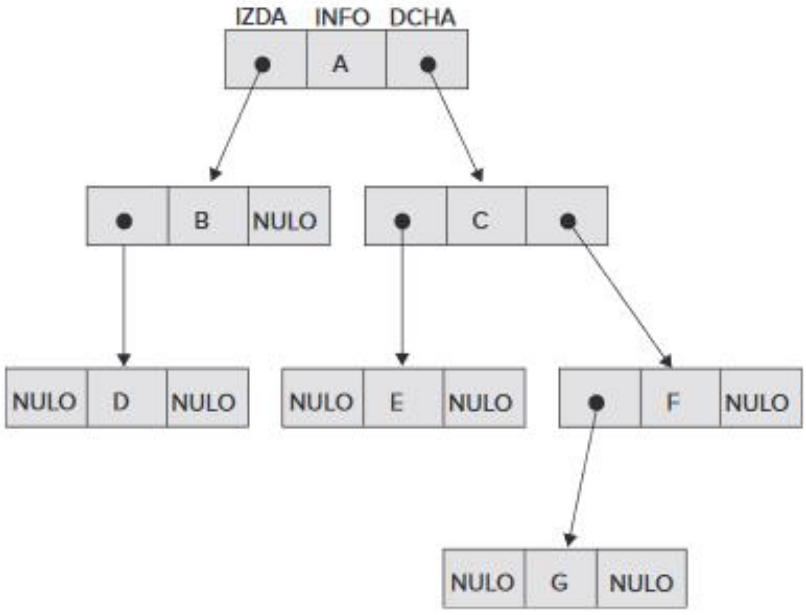


Figura 13.13. Árbol binario como lista enlazada.

Cristian Pilaparita 1095211-2202112336

13.3.4.3. Representación por arrays

Existen diferentes métodos, uno de los más fáciles es mediante tres arrays lineales paralelos que contemplan el campo de información y los dos punteros de ambos subárboles. Así, por ejemplo, el nodo raíz RAMÓN tendrá dos punteros IZQ:(9) –JOSEFINA– y DER:(16) –ANDRÉS–, mientras que el nodo CLEMENTE, al no tener descendientes, sus punteros se consideran cero (IZQ:0, DER:0).

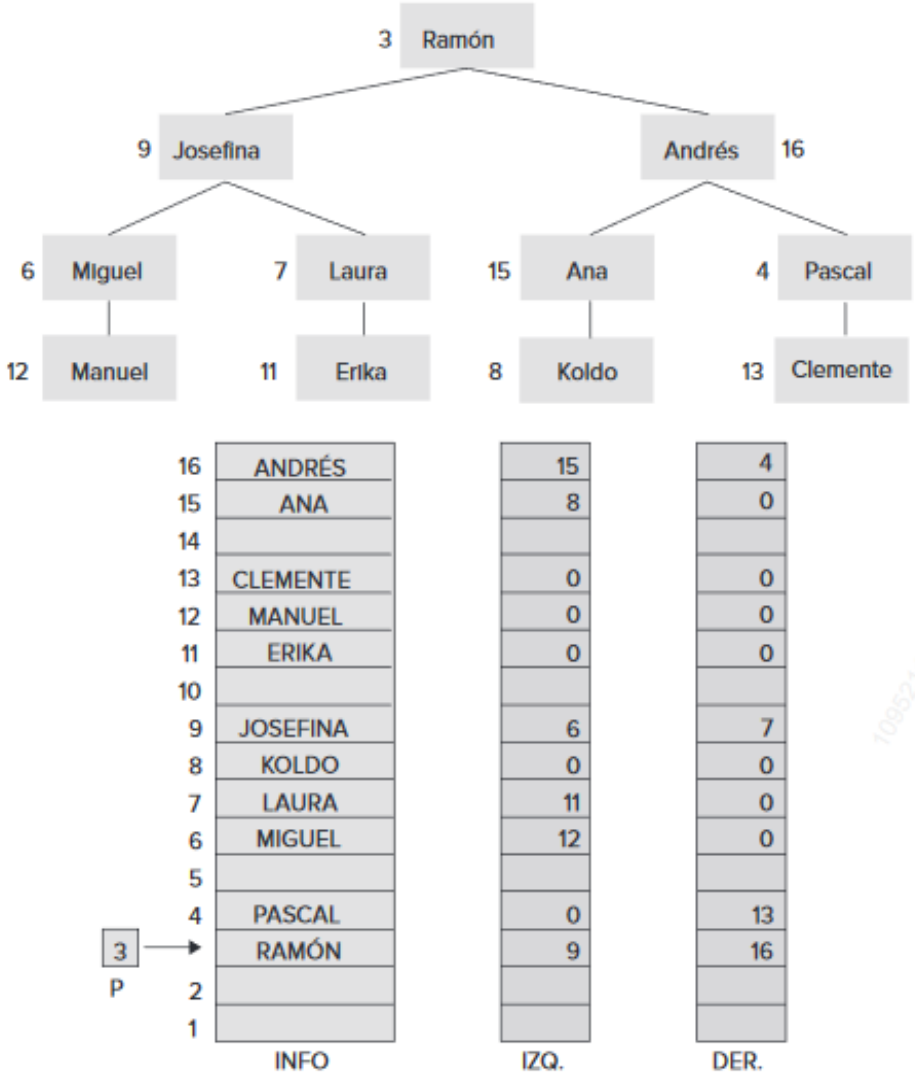
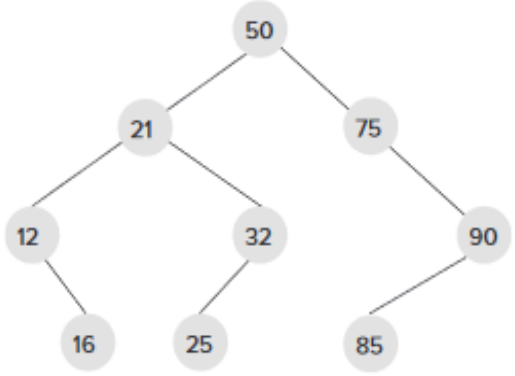


Figura 13.14. Árbol binario como arrays.

Otro método resulta más sencillo –un array lineal–. Para ello se selecciona un array lineal ÁRBOL.



Cristian Pilaparita 1095211-2202112336

El algoritmo de transformación es:

1. La raíz del árbol se guarda en ÁRBOL [1].
2. **si** un nodo *n* está en ÁRBOL[i] **entonces**
   su hijo izquierdo se pone en ÁRBOL[2*i]
   y su hijo derecho en ÁRBOL[2*i + 1]
   si un subárbol está vacío, se le da el valor NULO.

Este sistema requiere más posiciones de memoria que nodos tiene el árbol. Así, la transformación necesitará un array con 2^{h+2} elementos si el árbol tiene una profundidad *h*. En nuestro caso, como la profundidad es 3, requerirá 32 posiciones (2^5), aunque si no se incluyen las entradas nulas de los nodos terminales, veremos como sólo necesita catorce posiciones.

Árbol

1	50
2	21
3	75
4	12
5	32
6	0
7	90
8	0
9	16
10	25
11	
12	
13	
14	85
15	
16	
17	
18	
	. . .

Cristian Pilaparia 1095211-2202112336
1095211220211233

Un tercer método, muy similar al primero, sería la representación mediante un array de registros.

P = 3

	INFO	IZQ	DER
1			
2			
3	RAMÓN	9	16
4	PASCAL	0	13
5			
6	MIGUEL	12	0
7	LAURA	11	0
8	KOLDO	0	0
9	JOSEFINA	6	7
10			
11	ERIKA	0	0
12	MANUEL	0	0
13	CLEMENTE	0	0
14			
15	ANA	8	0
16	ANDRÉS	15	4

Cristian Pilapanta 1095211-2202112336

13.3.5. Recorrido de un árbol binario

Se denomina *recorrido de un árbol* el proceso que permite acceder una sola vez a cada uno de los nodos del árbol. Cuando un árbol se recorre, el conjunto completo de nodos se examina.

Existen muchos modos para recorrer un árbol binario. Por ejemplo, existen seis diferentes recorridos generales en profundidad de un árbol binario, simétricos dos a dos.

Los algoritmos de recorrido de un árbol binario presentan tres tipos de actividades comunes:

- *Visitar* el nodo raíz.
- *Recorrer* el subárbol izquierdo.
- *Recorrer* el subárbol derecho.

Estas tres acciones repartidas en diferentes órdenes proporcionan los diferentes recorridos del árbol en profundidad. Los más frecuentes tienen siempre en común recorrer primero el subárbol izquierdo y luego el subárbol derecho. Los algoritmos que lo realizan llaman *pre-orden, post-orden, in-orden* y su nombre refleja el momento en que se visita el nodo raíz. En el *in-orden* el raíz está en el medio del recorrido, en el *pre-orden* el raíz está al principio y en el *post-orden* está al último:

Recorrido pre-orden

1. Visitar el raíz.
2. Recorrer el subárbol izquierdo en *pre-orden*.
3. Recorrer el subárbol derecho en *pre-orden*.

Recorrido in-orden

1. Recorrer el subárbol izquierdo en *in-orden*.
2. Visitar el raíz.
3. Recorrer el subárbol derecho en *in-orden*.

Recorrido post-orden

1. Recorrer el subárbol izquierdo en *post-orden*.
2. Recorrer el subárbol derecho en *post-orden*.
3. Visitar el raíz.

Observe que todas estas definiciones tienen naturaleza recursiva.
En la figura 13.15 se muestran los recorridos de diferentes árboles binarios.

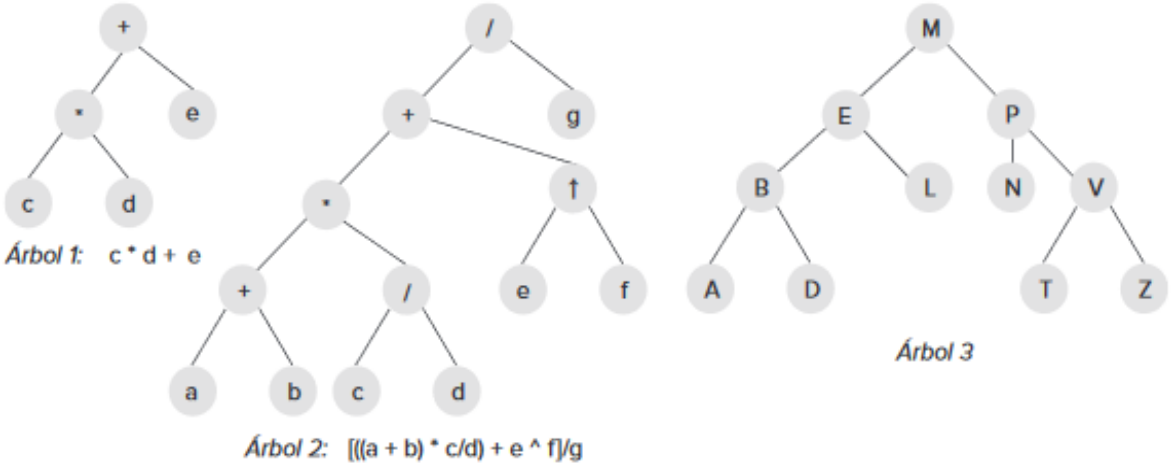


Figura 13.15. Recorrido de árboles binarios.

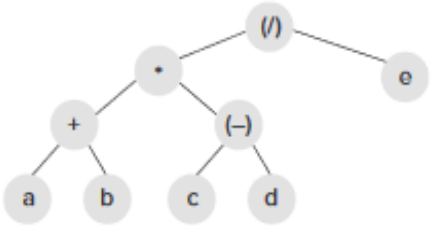
Árbol 1	*Pre-orden*	+ * c d e
	In-orden	c * d + e
	Post-orden	c d * e +
Árbol 2	*Pre-orden*	/ + * + a b / c d ^ e f g
	In-orden	a + b * c / d + e ^ f / g
	Post-orden	a b + c d / * e f ^ + g /
Árbol 3	*Pre-orden*	MEBADLPNVTZ
	In-orden	ABDELMNPTVZ
	Post-orden	ADBLENTZVPM

Cristian Pilaparita 1095211-2202112336

1095211220211233

EJEMPLO 13.4

Calcular los recorridos del árbol binario.

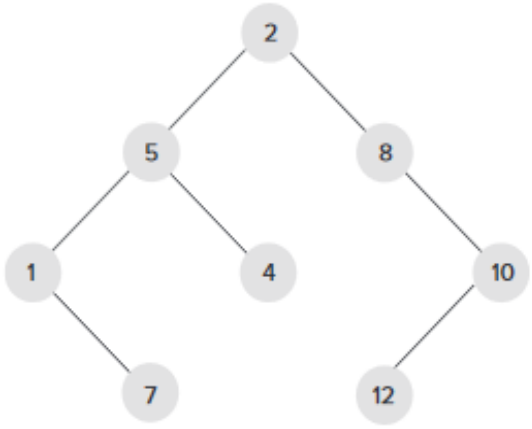


Solución

recorrido pre-orden	`/ * + ab - cde`
recorrido in-orden	`a + b * c - d/e`
recorrido post-orden	`ab + cd - * e/`

EJEMPLO 13.5

Realizar los recorridos del árbol binario.



Solución

recorrido pre-orden	2	5	1	7	4	8	10	12
recorrido in-orden	1	7	5	4	2	8	12	10
recorrido post-orden	7	1	4	5	12	10	8	2

13.4. Árbol binario de búsqueda

Recordará del capítulo 10, "Ordenación, búsqueda e intercalación", que para localizar un elemento en un array se podía realizar una búsqueda lineal; sin embargo, si el array era grande, una búsqueda lineal era ineficaz por su lentitud, especialmente si el elemento no estaba, ya que requería la lectura completa del array. Se ganaba tiempo si se clasificaba el array y se utilizaba una búsqueda binaria. Sin embargo, en un proceso de arrays las inserciones y eliminaciones son continuas, por lo que esto se hará complejo en cualquier método.

En los casos de gran número de operaciones sobre arrays o listas, lo que se necesita es una estructura donde los elementos puedan ser eficazmente localizados, insertados o borrados. Una solución a este problema es una variante del árbol binario que se conoce como *árbol binario de búsqueda* o *árbol binario clasificado* (*binary search tree*).

El árbol binario de búsqueda se construirá teniendo en cuenta las siguientes premisas:

- El primer elemento se utiliza para crear el nodo raíz.
- Los valores del árbol deben ser tales que pueda existir un orden (entero, real, lógico o carácter e incluso definido por el usuario si implica un orden).
- En cualquier nodo todos los valores del subárbol izquierdo del nodo son menores o iguales al valor del nodo. De modo similar, todos los valores del subárbol derecho deben ser mayores que los valores del nodo.

Si estas condiciones se mantienen, es sencillo probar que el recorrido *in-orden* del árbol produce los valores clasificados por orden. Así, por ejemplo, en la figura 13.16 se muestra un árbol binario.

Los tres recorridos del árbol son:

pre-orden	P	F	B	H	G	S	R	Y	T	W	Z
in-orden	B	F	G	H	P	R	S	T	Y	W	Z
post-orden	B	G	H	F	R	T	W	Z	Y	S	P

En esencia, un árbol binario contiene una clave en cada nodo que satisface las tres condiciones anteriores. Un árbol con las propiedades anteriores se denomina *árbol binario de búsqueda*.

Cristian Pilaparita 1095211-2202112336

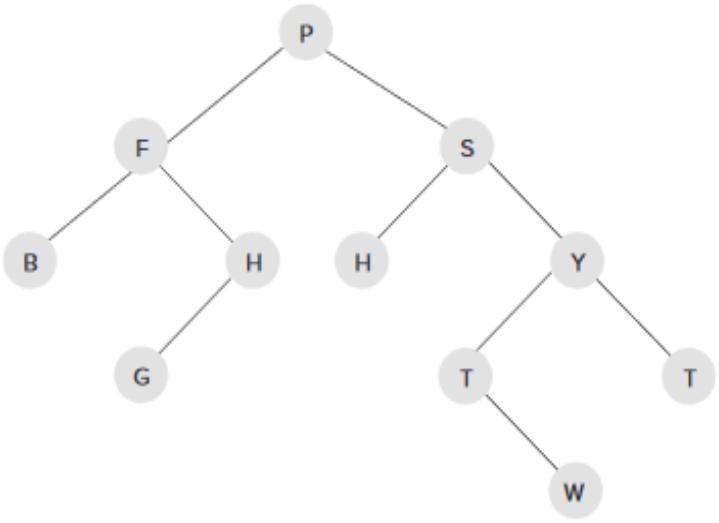


Figura 13.16. Árbol binario.

EJEMPLO 13.6

Se dispone de un array que contiene los siguientes caracteres:

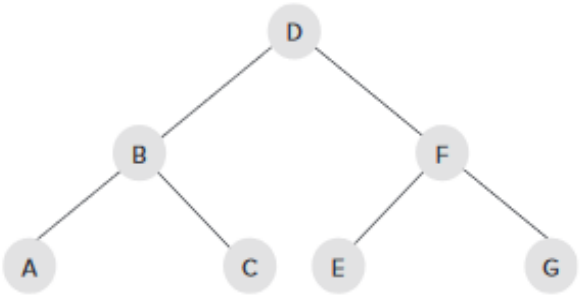
```
D F E B A C G
```

Construir un árbol binario de búsqueda.

Los pasos para la construcción del algoritmo son:

1. Nodo raíz del árbol: `D`.
2. El siguiente elemento se convierte en el descendente derecho, dado que `F` alfabéticamente es mayor que `D`.
3. A continuación, se compara `E` con el raíz. Dado que `E` es mayor que `D`, pasará a ser un hijo de `F` y como `E < F` será el hijo izquierdo.
4. El siguiente elemento `B` se compara con el raíz `D` y como `B < D` y es el primer elemento que cumple esta condición, `B` será el hijo izquierdo de `D`.
5. Se repiten los pasos hasta el último elemento.

El árbol binario de búsqueda resultante sería:



EJEMPLO 13.7

Construir el árbol binario de búsqueda correspondiente a la lista de números.

```
4  19  -7  49  100  0  22  12
```

El primer valor, como ya se ha comentado, es la raíz del árbol: es decir, 4. El siguiente valor, 19, se compara con 4; como es más grande se lleva al subárbol derecho de 4. El siguiente valor, –7, se compara con el raíz y es menor que su valor, 4; por tanto, se mueve al subárbol izquierdo. La figura 13.17 muestra los sucesivos pasos.

Cristian Pilapanta 1095211-2202112336

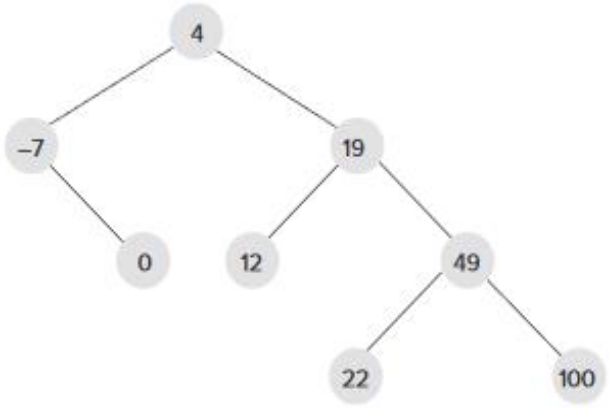
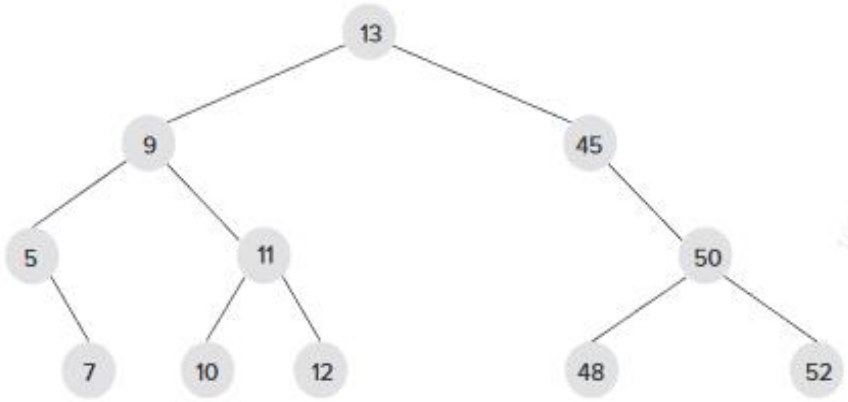


Figura 13.17. Construcción de un árbol binario.

13.4.1. Búsqueda de un elemento

La búsqueda en un árbol binario ordenado es dicotómica, ya que a cada examen de un nodo se elimina aquel de los subárboles que no contienen el valor buscado (valores todos inferiores o todos superiores).

Cristian Pilapanta 1095211-2202112336



El algoritmo de búsqueda del elemento —clave x— se realiza comparándolo con la clave del raíz del árbol. Si no es el mismo, se pasa al subárbol izquierdo o derecho, según el resultado de la comparación, y se repite la búsqueda en ese subárbol. La terminación del procedimiento se producirá cuando:

- Se encuentra la clave.
- No se encuentra la clave, se continúa hasta encontrar un subárbol vacío.

```
procedimiento buscar (E punt: RAÍZ;
                      E <tipo_elemento>: elemento;
                      S punt: actual, anterior)
   var
      lógico: encontrado
   inicio
      encontrado ← falso
      anterior ← nulo
      actual ← raíz
      mientras no encontrado Y (actual<>nulo) hacer
         si actual→.elemento = elemento entonces
            encontrado ← verdad
         si_no
            anterior ← actual
```

```
            si actual→.elemento > elemento entonces
               actual ← actual→.izdo
            si_no
               actual ← actual→.dcho
            fin_si
         fin_si
      fin_mientras
      si no encontrado entonces
         escribir('no existe', elemento)
      si_no
         escribir( elemento, 'existe')
      fin_si
   fin_procedimiento
   //< tipo_elemento> en este algoritmo es un tipo de dato simple
```

13.4.2. Insertar un elemento

Para insertar un elemento en el árbol A se ha de comprobar, en primer lugar, que el elemento no se encuentra en el árbol, ya que su caso no precisa ser insertado. Si el elemento no existe, la inserción se realiza en un nodo en el que al menos uno de los dos punteros `izq` o `der` tenga valor `nulo`.

Para realizar la condición anterior se desciende en el árbol a partir del nodo raíz, dirigiéndose de izquierda a derecha de un nodo, según que el valor a insertar sea inferior o superior al valor del campo clave `INFO` de este nodo. Cuando se alcanza un nodo del árbol en que no se puede continuar, el nuevo elemento se engancha a la izquierda o derecha de este nodo en función de que su valor sea inferior o superior al del nodo alcanzado.

El algoritmo de inserción del elemento `X` es:

```
procedimiento insertar (E/S punt: raíz;
                        E <tipo_elemento> : elemento)
var
   punt : nuevo,
          actual,
          anterior
inicio
   buscar (raíz, elemento, actual, anterior)
   si actual<> NULO entonces
        escribir ('elemento duplicado')
   si_no
        reservar (nuevo)
        nuevo→.elemento ← elemento
        nuevo→.izdo ← nulo
        nuevo→.dcho ← nulo
        si anterior = nulo entonces
           raíz ← nuevo
        si_no
           si anterior→.elemento > elemento entonces
              anterior→.izdo ← nuevo
           si_no
              anterior→.dcho ← nuevo
           fin_si
        fin_si
     fin_si
   fin_procedimiento
   // <tipo_elemento> es un tipo simple
```

Cristian Pitpanta 1095211-2202112336

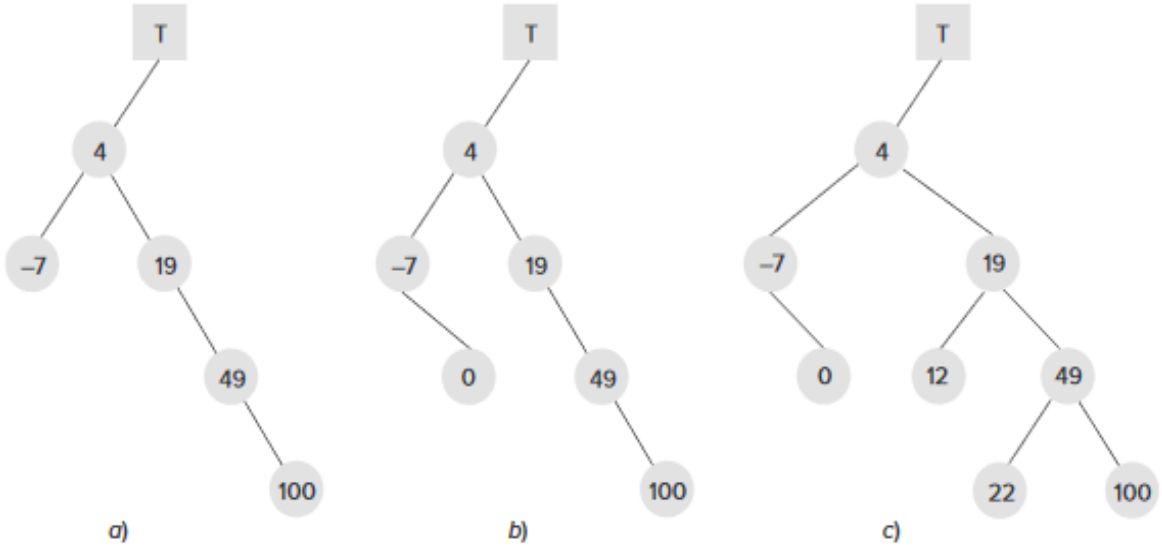


Figura 13.18. Inserciones en un árbol de búsqueda binaria: *a*) insertar 100, *b*) insertar (0), *c*) insertar 22 y 12.

Para insertar el valor `x` en el árbol binario ordenado se necesitará llamar al subprograma `insertar`.

Cristian Pilapanta 1095211-2202112336

13.4.3. Eliminación de un elemento

La eliminación de un elemento debe conservar el orden de los elementos del árbol. Se consideran diferentes casos, según la posición del elemento o nodo en el árbol:

- Si el elemento es una hoja, simplemente se suprime.
- Si el elemento no tiene más que un descendiente, se sustituye entonces por ese descendiente.
- Si el elemento tiene dos descendientes, se sustituye por el elemento inmediato inferior situado lo más a la derecha posible de su subárbol izquierdo.

Para poder realizar estas acciones será preciso conocer la siguiente información del nodo a eliminar:

- Conocer su posición en el árbol.
- Conocer la dirección de su padre.
- Conocer si el nodo a eliminar tiene hijos, si son 1 o 2 hijos, y en el caso de que sólo sea uno, si es hijo derecho o izquierdo.

La figura 13.19 muestra los tres posibles casos de eliminación de un nodo: *a*) eliminar `C`, *b*) eliminar `F`, *c*) eliminar `B`.

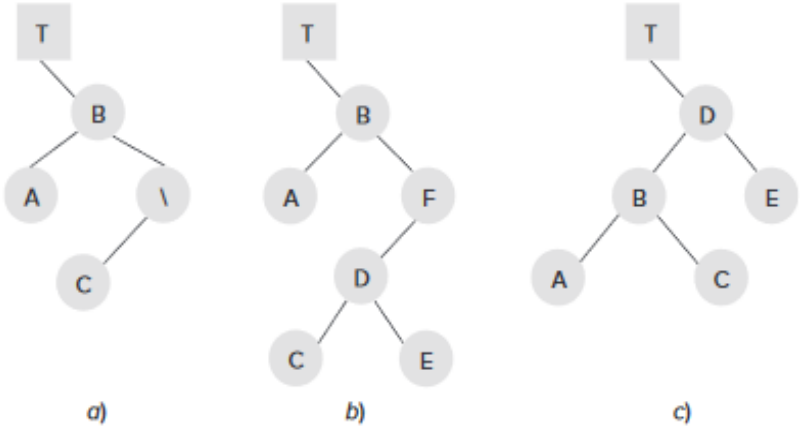


Figura 13.19. Casos posibles de eliminación de un nodo.

En la figura 13.20 se muestra el caso de eliminación de un nodo con un subárbol en un gráfico comparativo antes y después de la eliminación.

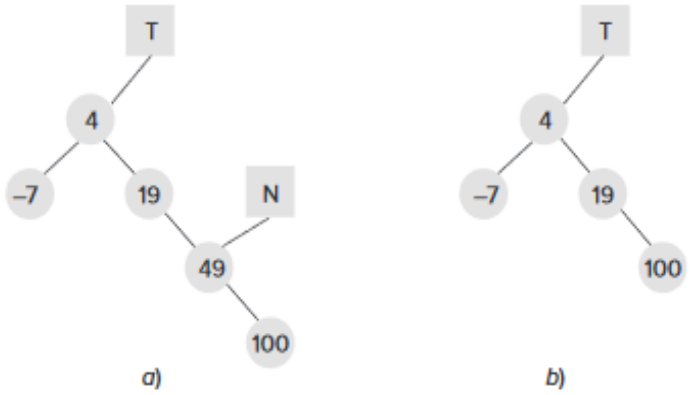


Figura 13.20. Eliminación de un nodo con un subárbol.

En la figura 13.21 se muestra el caso de la eliminación de un nodo (27) que tiene dos subárboles no nulos. En este caso se busca el nodo sucesor cuyo campo de información le siga en orden ascendente, es decir, 42, se intercambia entonces con el elemento que se desea borrar, 27.

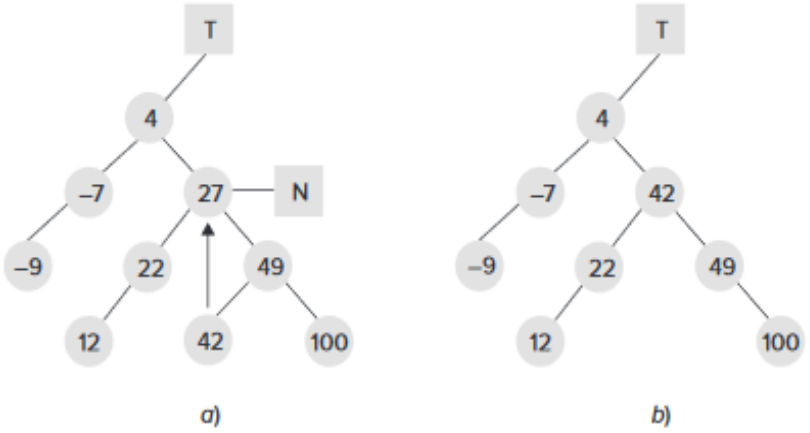
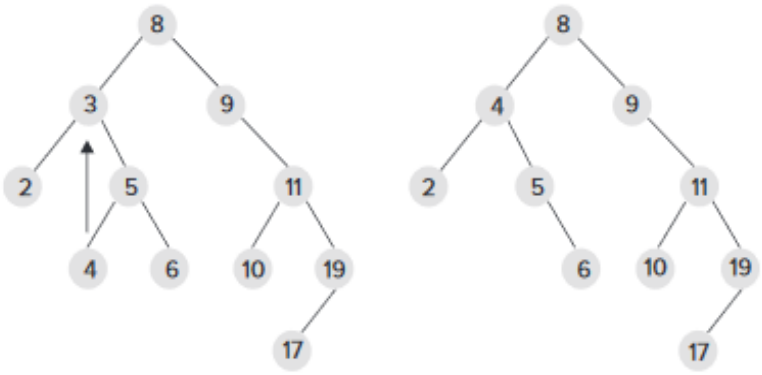


Figura 13.21. Eliminación de un nodo con dos subárboles no nulos.

EJEMPLO 13.8

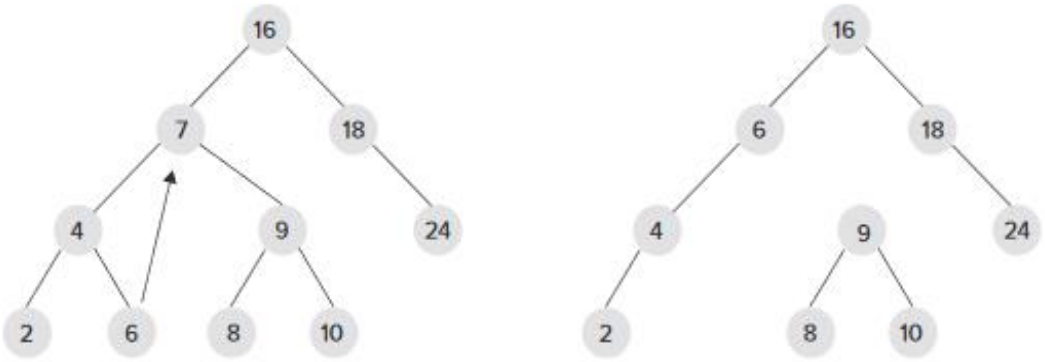
Deducir los árboles resultantes de eliminar el elemento 3 en el árbol A y el elemento 7 en el árbol B.

Árbol A



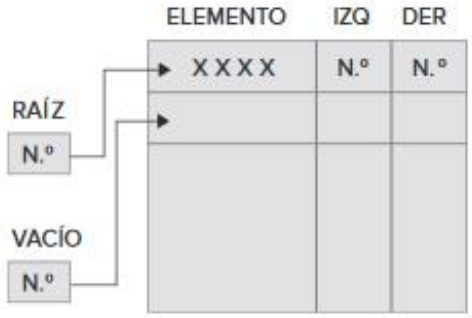
Cristian Pilapaña 1095211-2202112336

Árbol B. En este caso se busca el nodo sucesor cuyo campo de información le siga en orden decreciente, es decir, 6.



Árbol binario mediante arrays

Los árboles deben ser tratados como estructuras dinámicas; no obstante, si el lenguaje no tiene punteros podremos simularlos mediante arrays.



`Izdo` y `dcho` serán dos campos numéricos para indicar la posición en que están los hijos izquierdo y derecho. El valor 0 indicaría que no tiene hijo.

Trataremos el array como una lista `ENLAZADA` y necesitaremos una `LISTA DE VACÍOS` y una variable `VACÍO` que apunte al primer elemento de la lista de vacíos. Para almacenar la lista de vacíos es indiferente que utilicemos el campo `Izdo` o `Dcho`.

EJEMPLO 13.9

```
algoritmo árbol_binario_mediante_arrays
const
   Max = <expresión>
tipo
   registro: TipoElemento
      ... : ...
      ... : ...
   fin_registro
   registro: TipoNodo
      TipoElemento : Elemento
      Entero       : Izdo, Dcho
   fin_registro
   array[1..Max] de TipoNodo : Arr
```

Cristian Pilaparia 1095211-2202112336

```
var
   Arr          : a
   Entero       : opción
   TipoElemento : elemento
   Entero       : raíz
   Entero       : vacío
inicio
   iniciar(a,raíz,vacío)
   repetir
      menú
      escribir ('OPCIÓN: ')
      repetir
         leer (opción)
      hasta_que (opción >= 0) Y (opción <= 3)
      según_sea opción hacer
         1 :
              listado (a,raíz)
              escribir ('INTRODUZCA NUEVO ELEMENTO: ')
              proc_leer (elemento)
              altas (a, elemento, raíz, vacío)
              listado (a,raíz)
              pausa
         2 :
              listado (a,raíz)
              escribir ('INTRODUZCA ELEMENTO A DAR DE BAJA: ')
              proc_leer (elemento)
              bajas (a, elemento, raíz, vacío)
              listado (a,raíz)
              pausa
         3:
              listado (a,raíz)
              pausa
        fin_según hasta_que
     opción = 0
fin

procedimiento pausa
var
   cadena : c
inicio
   escribir('PULSE RETURN PARA CONTINUAR')
   leer(c)
fin_procedimiento

procedimiento menú
inicio
   escribir ('1.- ALTAS')
   escribir ('2.- BAJAS')
   escribir ('3.- LISTADO')
   escribir ('0.- FIN')
fin_procedimiento

procedimiento iniciar(S Arr: a;  S Entero: raíz, vacío)
var
   Entero: i
```

Cristian Pilaparia 1095211-2202112336
1095211220211233

```
inicio
   raíz ← 0
   vacío ← 1
   desde i ← 1 hasta Max-1 hacer
      a[i].dcho ← i+1
   fin_desde
   a[Max].dcho ← 0
fin_procedimiento

lógico función ÁrbolVacío(E Entero: raíz)
inicio
   si raíz=0 entonces
      devolver(verdad)
   si_no
      devolver(falso)
   fin_si
fin_función

lógico función ÁrbolLleno(E Entero: vacío)
inicio
   si vacío=0 entonces
      devolver(verdad)
   si_no
      devolver(falso)
   fin_si
fin_función

procedimiento inorden(E/S Arr: a;   E Entero: raíz)
inicio
   si raíz <> 0 entonces
      inorden(a,a[raíz].Izdo)
      proc_escribir(a[raíz].elemento)
      inorden(a,a[raíz].Dcho)
   fin_si
fin_procedimiento

procedimiento preorden(E/S Arr: a;   E Entero: raíz)
inicio
   si raíz <> 0 entonces
      proc_escribir(a[raíz].elemento)
      preorden(a, a[raíz].Izdo)
      preorden(a, a[raíz].Dcho)
   fin_si
fin_procedimiento

procedimiento postorden(E/S Arr: a;   E Entero: raíz)
inicio
   si raíz <> 0 entonces
      postorden(a,a[raíz].Izdo)
      postorden(a,a[raíz].Dcho)
      proc_escribir(a[raíz].elemento)
   fin_si
fin_procedimiento

procedimiento buscar(E/S Arr: a; E entero: raíz;
                     E TipoElemento: elemento;
                     S entero: act, ant)
```

Cristian Pilaparia 1095211-2202112337
1095211220211233

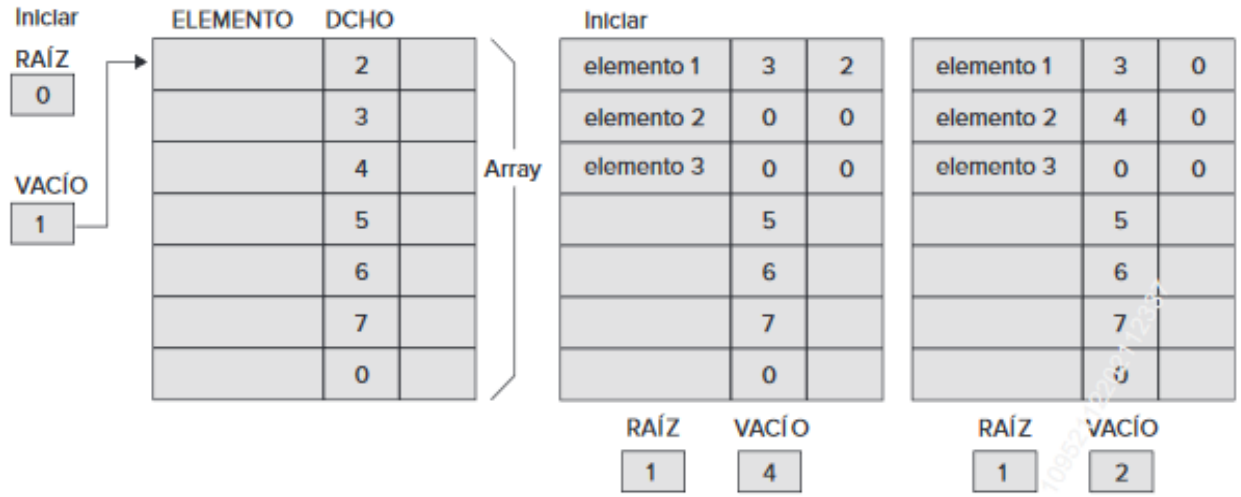
```
var
   lógico: encontrado
inicio
   encontrado ← falso
   act ← raíz
   ant ← 0
   mientras no encontrado y (act<>0) hacer
      si igual(elemento, a[act].elemento) entonces
         encontrado ← verdad
      si_no
         ant ← act
         si mayor(a[act].elemento, elemento) entonces
            act ← a[act].Izdo
         si_no
            act ← a[act].Dcho
        fin_si
      fin_si
   fin_mientras
fin_procedimiento

procedimiento altas(E/S Arr: A; E TipoElemento: elemento;
                    E/S entero: raíz, vacío)
var
   entero: act,ant,auxi
inicio
   si vacío <> 0 entonces
      buscar(a, raíz, elemento, act, ant)
      si act <> 0 entonces
         escribir('ESE ELEMENTO YA EXISTE')
      si_no
         auxi ← vacío
         vacío ← a[auxi].Dcho
         a[auxi].elemento ← elemento
         a[auxi].Izdo ← 0
         a[auxi].Dcho ← 0
         si ant = 0 entonces
            raíz ← auxi
         si_no
            si mayor(a[ant].elemento, elemento) entonces
               a[ant].Izdo ← auxi
            si_no
               a[ant].Dcho ← auxi
            fin_si
         fin_si
      fin_si
   fin_si
fin_procedimiento

procedimiento bajas(E/S Arr: A; E TipoElemento: elemento;
                    E/S entero: raíz, vacío)
var
   entero: act, ant, auxi
inicio
   buscar(a, raíz, elemento, act, ant)
   si act = 0 entonces
```

Cristian Pliparta 1095211-2202112337

```
      escribir('ESE ELEMENTO NO EXISTE')
   si_no
      si (a[act].Izdo = 0) Y (a[act].Dcho = 0) entonces
         si ant = 0 entonces
            raíz ← 0
         si_no
            si a[ant].Izdo = act entonces
               a[ant].Izdo ← 0
            si_no
               a[ant].Dcho ← 0
            fin_si
         fin_si
      si_no
         si (a[act].Izdo <> 0) Y (a[act].Dcho <> 0) entonces
            ant ← act
            auxi ← a[act].Izdo
            mientras a[auxi].Dcho <> 0 hacer
               ant ← auxi
               auxi ← a[auxi].Dcho
            fin_mientras
            a[act].Elemento ← a[auxi].Elemento
            si ant = act entonces
               a[ant].Izdo ← a[auxi].Izdo
            si_no
               a[ant].Dcho ← a[auxi].Izdo
            fin_si
            act ← auxi
         si_no
            si a[act].Dcho <> 0 entonces
               si ant = 0 entonces
                  raíz ← a[act].Dcho
               si_no
                  si a[ant].Izdo = act entonces
                     a[ant].Izdo ← a[act].Dcho
                  si_no
                     a[ant].Dcho ← a[act].Dcho
                  fin_si
               fin_si
            si_no
               si ant = 0 entonces
                  raíz ← a[act].Izdo
               si_no
                  si a[ant].Dcho = act entonces
                     a[ant].Dcho ← a[act].Izdo
                  si_no
                     a[ant].Izdo ← a[act].Izdo
                  fin_si
               fin_si
            fin_si
         fin_si
      fin_si
      a[act].Dcho ← vacío
```

Cristian Pilapanta 1095211-2202112337

```
      vacío ← act
   fin_si
fin_procedimiento

procedimiento listado (E Arr: a;   E Entero: raíz)
inicio
   escribir ('INORDEN: ')
   inorden (a, raíz)
   escribir ('PREORDEN: ')
   preorden (a, raíz)
   escribir ('POSTORDEN: ')
   postorden (a, raíz)
fin_procedimiento
```



13.5. Grafos

Los grafos son otra estructura de datos no lineal y que tiene gran número de aplicaciones. El estudio del análisis de grafos ha interesado a los matemáticos durante siglos y representa una parte importante de la teoría combinatoria en matemáticas. Aunque la teoría de grafos es compleja y amplia, en esta sección se realizará una introducción a la teoría de grafos y a los algoritmos que permiten su solución por computadora.

Los árboles binarios representan estructuras jerárquicas con limitaciones de dos subárboles por cada nodo. Si se eliminan las restricciones de que cada nodo puede apuntar a dos nodos —como máximo— y que cada uno puede estar apuntado por otro nodo —como máximo— nos encontramos con un grafo.

Ejemplos de grafos en la vida real los tenemos en la red de carreteras de un estado o región, la red de enlaces ferroviarios o aéreos nacionales, etcétera.

En una red de carreteras los nudos de la red representan los *vértices* del grafo y las carreteras de unión de dos ciudades los *arcos*, de modo que a cada arco se asocia una información tal como la distancia, el consumo en gasolina por automóvil, entre otros.

Los grafos nos pueden ayudar a resolver problemas como éste. Suponga que ciertas carreteras del norte del estado han sido bloqueadas por una reciente tormenta de nieve. ¿Cómo se puede saber si todas las ciudades de ese estado se pueden alcanzar por carretera desde la capital o si existen ciudades aisladas? Evidentemente existe la solución del estudio de un mapa de carreteras; sin embargo, si existen muchas ciudades, la obtención de la solución puede ser ardua y costosa en tiempo. Una computadora y un algoritmo adecuado de grafos solucionarán fácilmente el problema.

Cristian Pilapaña 1095211-2202112337

13.5.1. Terminología de grafos

Formalmente un *grafo* es un conjunto de puntos —una estructura de datos— y un conjunto de líneas, cada una de las cuales une un punto a otro. Los puntos se llaman *nodos* o *vértices* del grafo y las líneas se llaman *aristas* o *arcos* (*edges*).

Se representan el conjunto de vértices de un grafo dado G por V_G y el conjunto de arcos por A_G. Por ejemplo, en el grafo G de la figura 13.23:

V_G = {a, b, c, d}

A_G = {1, 2, 3, 4, 5, 6, 7, 8}

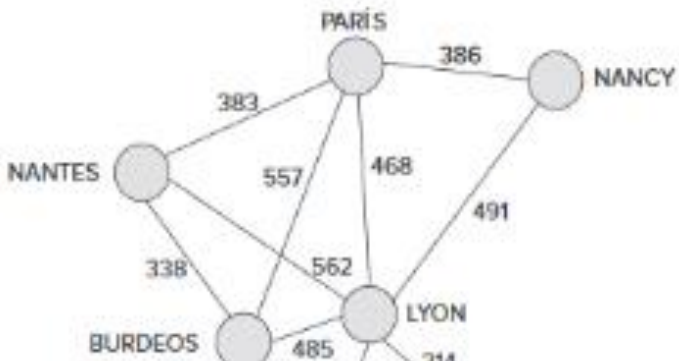


Figura 13.22. Grafo de una red de carreteras.

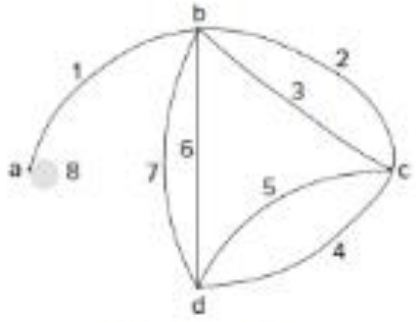


Figura 13.23. Grafo G.

El número de elementos de V_G se llama *orden* del grafo. Un *grafo nulo* es un grafo de orden cero.

Una arista se representa por los vértices que conecta; la arista 3 conecta los vértices b y c, y se representa por *V(b, c)*. Algunos vértices pueden conectar un nodo consigo mismo; por ejemplo, la arista 8 tiene el formato *V(a, a)*. Estas aristas se denominan *bucles* o *lazos*.

Un grafo G se denomina *sencillo* si se cumplen las siguientes condiciones:

- No tiene lazos, no existe un arco en A_G de la forma (*V*, *V*).
- No existe más que un arco para unir dos nodos, es decir, no existe más que un arco (V_1, V_2) para cualquier par de vértices V_1, V_2.

Cristian Pilaparita 1095211-2202112337

En la figura 13.24 se representa un grafo sencillo.

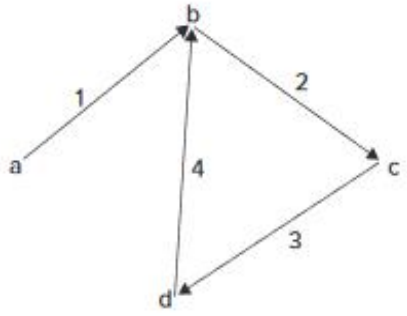


Figura 13.24. Grafo sencillo.

Un grafo que no es sencillo se denomina *grafo múltiple*.

Un *camino* es una secuencia de uno o más arcos que conectan dos nodos. Representaremos por $C(V_i, V_j)$ un camino que conecta los vértices V_i y V_j.

La *longitud* de un camino es el número de arcos que comprende. En el grafo de la figura 13.24 existen los siguientes caminos entre los nodos *b* y *d*.

```
C(b,d) = (b,c) (c,d)                    longitud = 2
C(b,d) = (b,c) (c,b) (b,c) (c,d)        longitud = 4
C(b,d) = (b,d)                          longitud = 1
C(b,d) = (b,d) (c,b) (b,d)              longitud = 3
```

Se dice que dos vértices son *adyacentes* (inmediatos) si hay un arco que los une; así, V_i y V_j son adyacentes si existe un camino que los une. Esta definición es muy general y normalmente se particulariza, si existe un camino desde *A* a *B*, decimos que *A es adyacente a B* y *B es adyacente desde A*; así, en el grafo de la figura 13.25, Las Vegas es adyacente a Nueva York, pero Nueva York no es adyacente a Las Vegas.

Se consideran dos tipos de grafos:

Dirigidos los vértices apuntan unos a otros, los arcos están dirigidos o tienen dirección.

No-dirigidos los vértices están relacionados, pero no se apuntan unos a otros; la dirección no es importante.

En la figura 13.25 el grafo es dirigido, dado que la dirección es importante; así, existe un vuelo entre Las Vegas y Nueva York, pero no en sentido contrario.

grafo conectado existe siempre un camino que une dos vértices cualesquiera,

grafo desconectado existen vértices que no están unidos por un camino.

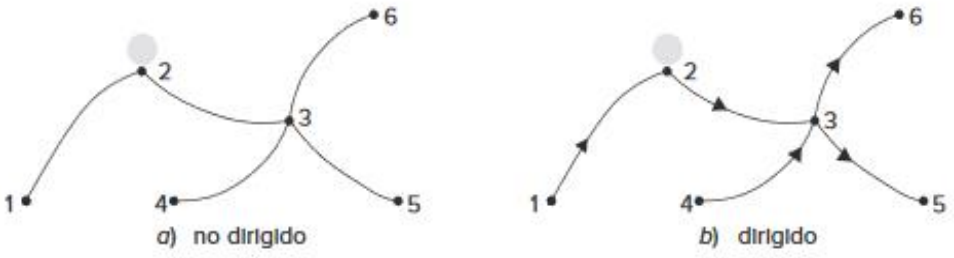
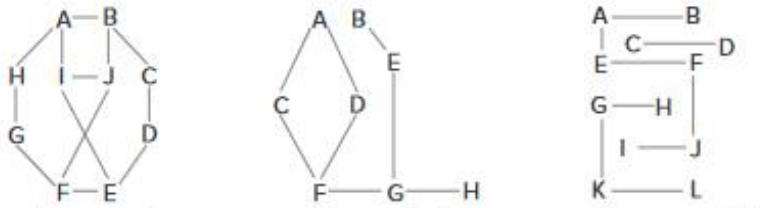


Figura 13.25. Grafo: *a*) no dirigido, *b*) dirigido.

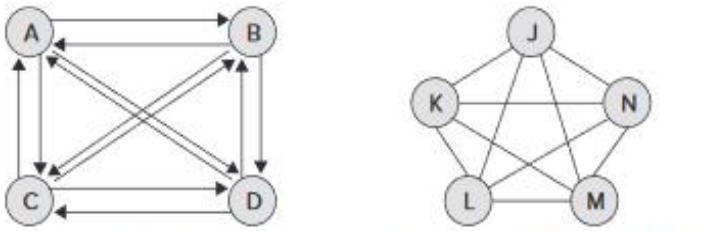
Cristian Pilapanta 1095211-2202112337



a) conectado *b*) conectado *b*) no conectado

Figura 13.26. Grafos conectados y no conectados.

Otros tipos de grafos de gran interés se muestran en la figura 13.27. Un *grafo completo* es aquel en que cada vértice está conectado con todos y cada uno de los restantes nodos. Si existen n vértices, habrá $n(n-1)$ arcos en un grafo completo y dirigido, y $n(n-1)/2$ aristas en un grafo no dirigido completo.



a) grafo completo dirigido *b*) grafo completo no dirigido

Figura 13.27. Grafos completos.

Un *grafo ponderado o con peso* es aquel en el que cada arista o arco tiene un valor. Los grafos con peso suelen ser muy importantes, ya que pueden representar situaciones de gran interés; por ejemplo, los vértices pueden ser ciudades y las aristas distancias o precios del pasaje de ferrocarril o avión entre ambas ciudades. Eso nos puede permitir calcular cuál es el recorrido más económico entre dos ciudades, sumando los importes de los billetes de las ciudades existentes en el camino y así poder tomar una decisión acertada respecto al viaje e incluso estudiar el posible cambio de medio de transporte: avión o automóvil, si éstos resultan más baratos.

La solución de encontrar el camino más corto, el de menor precio o más económico entre dos vértices de un grafo, es un algoritmo importante en la teoría de grafos. (El *algoritmo de Dijkstra* es un tipo para la solución de dichos problemas.)

13.5.2. Representación de grafos

Existen dos técnicas estándar o apuntadores para representar un grafo G: *la matriz de adyacencia* (mediante arrays) y la *lista de adyacencia* (mediante punteros/listas enlazadas).

13.5.2.1. Matriz de adyacencia

La matriz de adyacencia `M` es un array de dos dimensiones que representa las conexiones entre pares de vértices. Sea un grafo `G` con un conjunto de nodos V_G y un conjunto de aristas A_G. Supongamos que el grafo es de orden `N`, donde `N` >= 1. La matriz de adyacencia `M` se representa por una matriz de `N` × `N` elementos, donde:

$$M(i, j) = \begin{cases} 1 \text{ si existe un arco } (V_i, V_j) \text{ en } A_G, V_i \text{ es adyacente a } V_j \\ 0, \text{ en caso contrario} \end{cases}$$

Cristian Pilapaña 1095211-2202112337

Las columnas y las filas de la matriz representan los vértices del grafo. Si existe una arista desde *i* a *j* (esto es, el vértice *i* es adyacente a *j*), se introduce un 1; si no existe la arista, se introduce un 0; lógicamente, los elementos de la diagonal principal son todos ceros, ya que el coste de la arista *i* a *i* es 0.

Si *G* es un grafo no dirigido, la matriz es simétrica *M*(*i*, *j*) = *M*(*j*, *i*). La matriz de adyacencia del grafo de la figura 13.25 se indica en la figura 13.28.

i \ j	1	2	3	4	5	6
1	0	1	0	0	0	0
2	1	0	1	0	0	0
3	0	1	0	1	1	1
4	0	0	1	0	0	0
5	0	0	1	0	0	0
6	0	0	1	0	0	0

Figura 13.28. Matriz de adyacencia.

Si el grafo fuese dirigido, su matriz resultante sería:

i \ j	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	1	0	0	0
3	0	0	0	0	1	1
4	0	0	1	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0

Cristian Pilapanta 1095211-2202112337

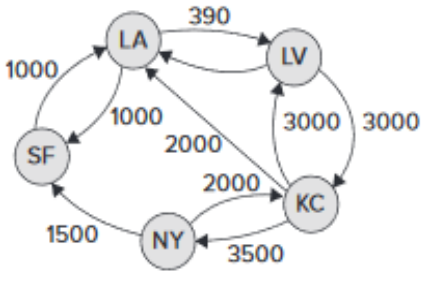
EJEMPLO 13.10

Deducir la matriz de adyacencia del grafo siguiente:

La matriz de adyacencia resultante de este grafo, cuyos vértices representan ciudades y los pesos de las aristas, los precios de pasajes de avión en dólares es:

Solución

	SF	LA	LV	KC	NY
SF		1000			
LA	1000		390	2000	
LV		390		3000	2500
KC		2000	3000		
NY	1500			350	



EJEMPLO 13.11

Sea un grafo con aristas ponderadas. Los vértices representan ciudades y las aristas las rutas utilizadas por los camiones de una empresa de transporte de mercancías. Cada arista está rotulada con la distancia entre las parejas de ciudades enlazadas directamente. En este caso utilizaremos una matriz triangular, ya que la matriz es simétrica.

Nota

Observe como consecuencia de este ejemplo que las aristas ponderadas tienen una gran aplicación.

- En transporte comúnmente representan distancias, precios de billetes, tiempos.
- En hidráulica, capacidades. Por ejemplo, el caudal de un oleoducto entre diferentes ciudades litros/segundo.

13.5.2.2. Lista de adyacencia

El segundo método utilizado para representar grafos es útil cuando un grafo tiene muchos vértices y pocas aristas; es la *lista de adyacencia*. En esta representación se utiliza una lista enlazada por cada vértice *v* del grafo que tenga vértices adyacentes *desde* él.

El grafo completo incluye dos partes: un directorio y un conjunto de listas enlazadas. Hay una entrada en el directorio por cada nodo del grafo. La entrada en el directorio del nodo *i* apunta a una lista enlazada que representa los nodos que son conectados al nodo *i*. Cada registro de la lista enlazada tiene dos campos: uno es un identificador de nodo, otro es un enlace al siguiente elemento de la lista; la lista enlazada representa arcos.

Una lista de adyacencia del grafo de la figura 13.25a se da en la figura 13.29.

Cristian Pilaparita 1095211-2202112341

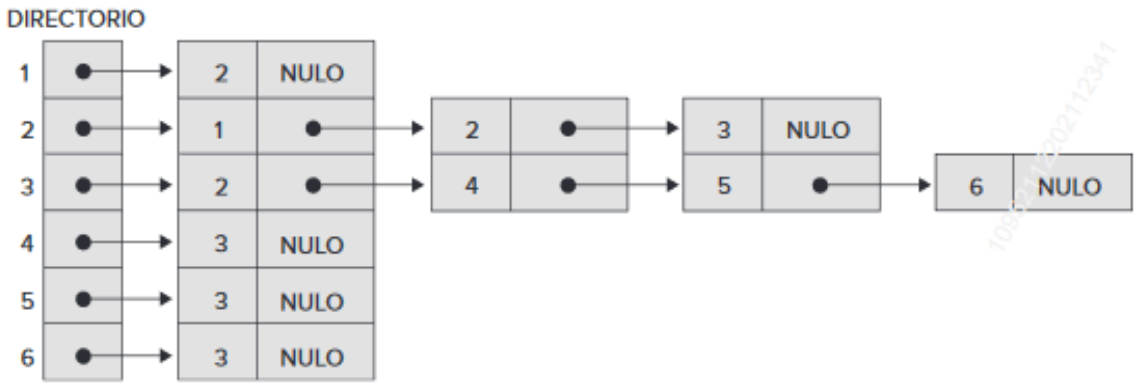


Figura 13.29. Lista de adyacencia.

Un grafo no dirigido de orden `N` con `A` arcos requiere `N` entradas en el directorio y `2*A` entradas de listas enlazadas, excepto si existen bucles que reducen el número de listas enlazadas en 1.

Un grafo dirigido de orden `N` con `A` arcos requiere `N` entradas en el directorio y `A` entradas de listas enlazadas.

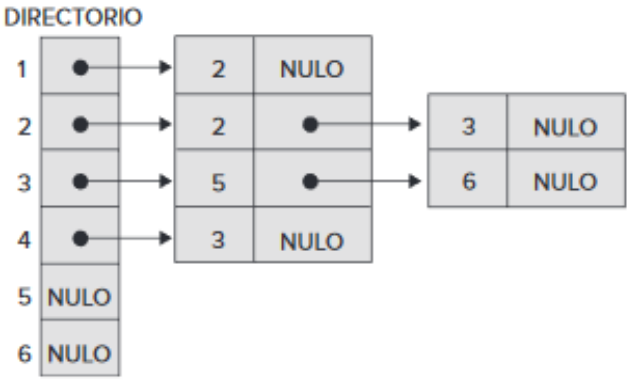
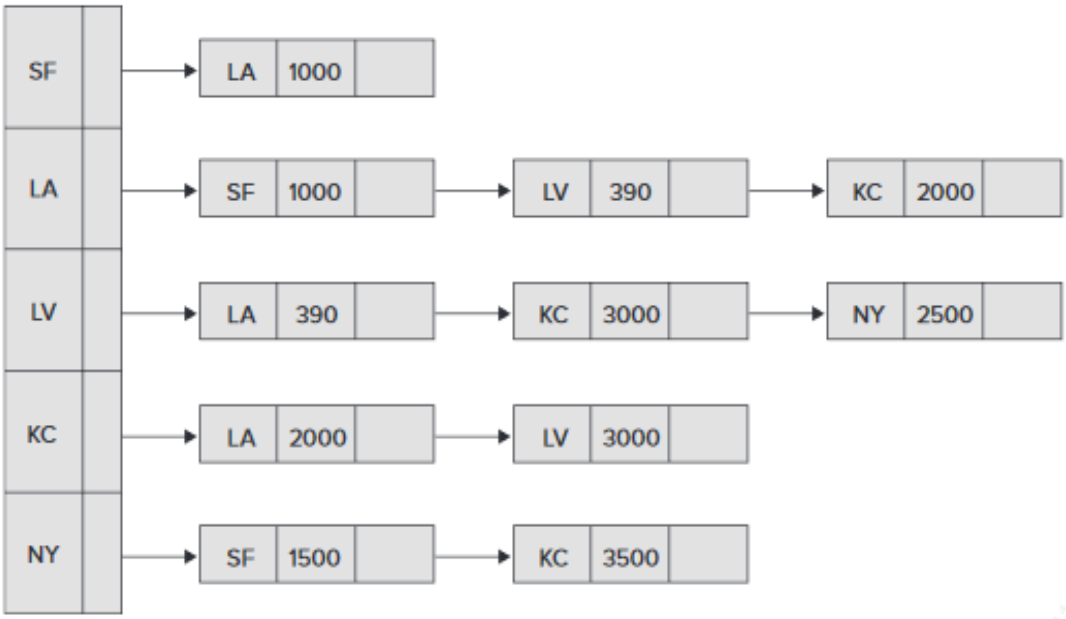


Figura 13.30.

EJEMPLO 13.12

La lista de adyacencia del grafo del ejemplo 13.10 es:

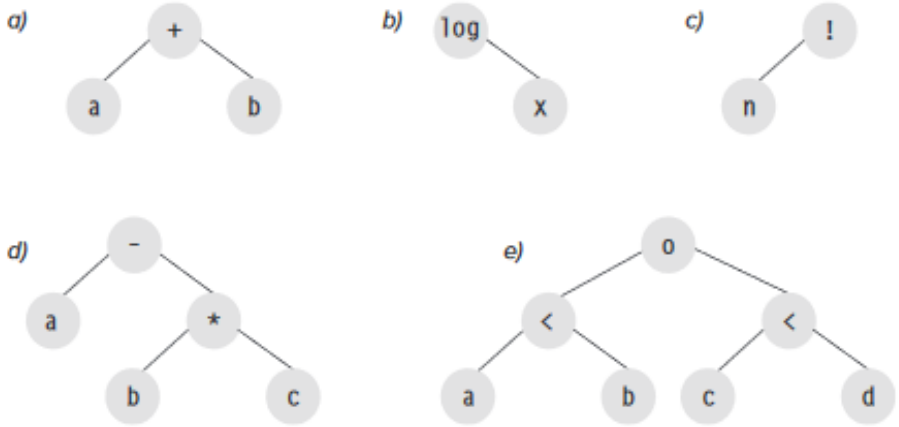


La elección de la representación depende del algoritmo particular que se vaya a implementar y si el grafo es "disperso" o "denso". Un grafo disperso es uno en el que el número de vértices N es mucho mayor que el número de arcos. En un grafo denso el número de arcos se acerca al máximo.

Cristian Pilapanta 1095211-2202112341

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

13.1. *Deducir las fórmulas de las expresiones representadas por los siguientes árboles de expresión.*



a) $a + b$

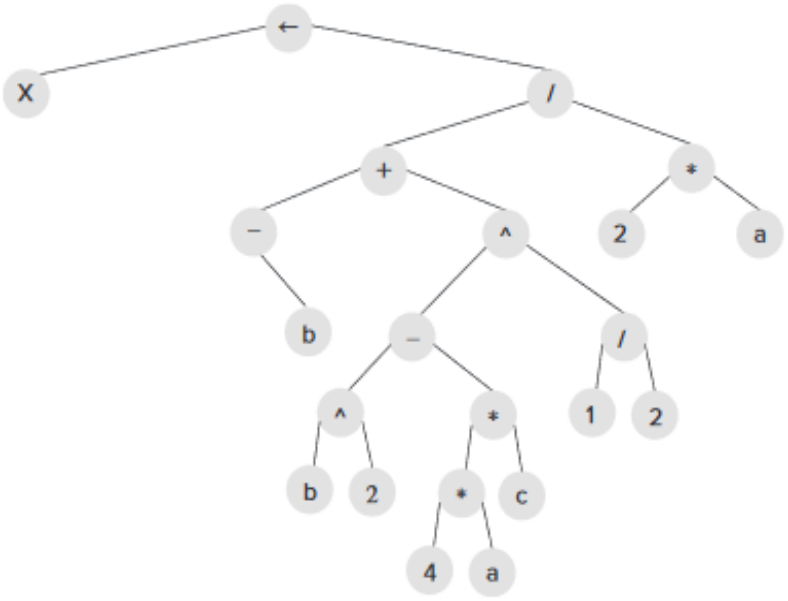
b) $\log x$

c) $n!$

d) $a - (b * c)$

e) $(a < b)$ o $(c < d)$

13.2. *Deducir la fórmula que representa el siguiente árbol expresión.*



$x = (-b + (b^2 - 4 * a * c)^{(1/2)}) / (2 * a)$

es decir, una de las raíces solución de la ecuación cuadrática o de segundo grado:

$ax^2 + bx + c = 0.$

13.3. *Teniendo en cuenta que nuestro lenguaje de programación no maneja estructuras dinámicas de datos, escribir un procedimiento que inserte un nuevo nodo en un árbol binario en el lugar correspondiente según su valor. Escribir otro procedimiento que permita conocer el número de nodos de un árbol binario. Utilizar ambos procedimientos desde un algoritmo que cree el árbol y nos informe sobre su número de nodos.*

Análisis del problema

El procedimiento de inserción será análogo al de altas que aparece en el ejercicio 13.9. Para conocer el número de nodos del árbol se realiza su recorrido, por uno cualquiera de los métodos ya comentados —inorden, preorden, postorden— y se irán contando.

El programa principal comenzará con un proceso de inicialización, a continuación utilizará una estructura repetitiva que permita la inserción de un número indeterminado de nodos en el árbol y, por último, llamará al procedimiento para contarlos.

Al no especificarse en el enunciado el tipo de información que se almacena en los registros del árbol, ésta se tratará de forma genérica, recurriendo a procedimientos y funciones auxiliares no desarrollados que permitan manipularla, como, por ejemplo, `leerelemento(elemento)`, `escribirelemento(elemento)`, `distinto(elemento,'0')`.

Diseño del algoritmo

```
algoritmo ejercicio_13_3
const
   Máx = ...
```

Cristian Pilapanta 1095211-2202112341

```
tipo
   registro: tipoelemento
      ... : ...
      ... : ...
   fin_registro
   registro: tiponodo
      tipoelemento : elemento
      entero       : izdo, dcho
   fin_registro
   array[1..Máx] de tiponodo: arr
var
   arr          : a
   tipoelemento : elemento
   entero       : raíz, vacío
inicio
   iniciar(a,raíz,vacío)
   escribir ('Introduzca nuevo elemento: ')
   si no árbollleno(vacío) entonces
      leerelemento (elemento)
   fin_si
   mientras distinto(elemento,'0') y no árbollleno(vacío) hacer
      altas (a, elemento, raíz, vacío)
      si no árbollleno(vacío) entonces
         escribir ('Introduzca nuevo elemento: ')
         leerelemento (elemento)
      fin_si
   fin_mientras
   listado (a, raíz)
fin

procedimiento iniciar(S arr: a  ;   S entero: raíz, vacío)
var
   entero: i
inicio
   raíz ← 0
   vacío ← 1
   desde i ← 1 hasta Máx-1 hacer
      a[i].dcho ← i+1
   fin_desde
   a[Máx].dcho ← 0
fin_procedimiento
lógico función árbollleno(E entero: vacío)
inicio
   si vacío = 0 entonces
      devolver(verdad)
   si_no
      devolver(falso)
   fin_si
fin_función

procedimiento inorden(E arr: a;  E entero: raíz;  E/S entero: cont)
inicio
   si raíz <> 0 entonces
```

Cristian Pilapanta 1095211-2202112341
1095211220211234

```
      inorden(a, a[raíz].izdo, cont)
      cont ← cont + 1
      escribirelemento(a[raíz].elemento)
      // Además de contar los nodos visualiza la
      // información almacenada en ellos
      inorden(a, a[raíz].dcho, cont)
   fin_si
fin_procedimiento

procedimiento buscar(E arr: a ; E tipoelemento:elemento;
                     S entero: act, ant)
var
   lógico: encontrado
inicio
   encontrado ← falso
   act ← raíz
   ant ← 0
   mientras no encontrado y (act <> 0) hacer
      si igual(elemento, a[act].elemento) entonces
         encontrado ← verdad
      si_no
         ant ← act
         si mayor(a[act].elemento, elemento) entonces
            act ← a[act].izdo
         si_no
            act ← a[act].dcho
         fin_si
      fin_si
   fin_mientras
fin_procedimiento

procedimiento altas(E/S arr: a;  E tipoelemento: elemento;
                    E/S entero: raíz, vacío)
var
   entero: act, ant, auxi
inicio
   si no árbollleno(vacío) entonces
      buscar(a, elemento, act, ant)
      si act <> 0 entonces
         escribir('Ese elemento ya existe')
      si_no
         auxi ← vacío
         vacío ← a[auxi].dcho
         a[auxi].elemento ← elemento
         a[auxi].izdo ← 0
         a[auxi].dcho ← 0
         si ant = 0 entonces
            raíz ← auxi
         si_no
            si mayor(a[ant].elemento, elemento) entonces
               a[ant].izdo ← auxi
            si_no
               a[ant].dcho ← auxi
            fin_si
```

Cristian Pilapaña 1095211-2202112341
1095211220211234

```
            fin_si
         fin_si
      fin_si
fin_procedimiento

procedimiento listado (E arr: a; E entero: raíz)
var
   entero: cont
inicio
   escribir ('inorden: ')
   cont ← 0
   inorden (a, raíz, cont)
   escribir('El número de nodos es ', cont)
fin_procedimiento
```

13.4. *Escribir un procedimiento que permita contar las hojas de un árbol mediante estructuras dinámicas.*

Análisis del problema

Recorrer el árbol contando únicamente los nodos que no tienen hijos.

Diseño del algoritmo

```
procedimiento contarhojas(E punt: raíz;  E/S entero: cont)
inicio
   si raíz <> nulo entonces
      si (raíz→.izdo = nulo) y (raíz→.dcho = nulo) entonces
         cont ← cont+1
      fin_si
      contarhojas(raíz→.izdo, cont)
      contarhojas(raíz→.dcho, cont)
   fin_si
fin_procedimiento
```

13.5. *Diseñar una función que permita comprobar si son iguales dos árboles cuyos nodos tienen la siguiente estructura:*

```
tipo
   puntero_a nodo: punt
   registro  : nodo
      entero : elemento
      punt   : izdo, dcho
   fin_registro
```

Análisis del problema

Se trata de una función recursiva que compara nodo a nodo la información almacenada en ambos árboles. Las condiciones de salida del proceso recursivo serán que:

- Se termine de recorrer uno de los dos árboles.
- Se terminen de recorrer ambos.
- Se encuentre diferente información en los nodos comparados.

Si los árboles terminaron de recorrerse simultáneamente es que ambos tienen el mismo número de nodos y nunca ha sido diferente la información comparada; por tanto, la función devolverá verdad; en cualquier otro caso la función devolverá falso.

Diseño del algoritmo

```
lógico función iguales(E punt: raíz1, raíz2)
inicio
```

Cristian Pilaparia 1095211-2202112341

```
    si raíz1 = nulo entonces
       si raíz2 = nulo entonces
          devolver(verdad)
       fin_si
    si_no
       si raíz2 = nulo entonces
          devolver(falso)
       si_no
          si raíz1→.elemento <> raíz2→.elemento entonces
             devolver(falso)
          si_no
             devolver(iguales(raíz1→.izdo, raíz2→.izdo)
                y iguales(raíz1→.dcho, raíz2→.dcho))
          fin_si
       fin_si
    fin_si
fin_función
```

Cristian Pilaparia 1095211-2202112341

CONCEPTOS CLAVE

- Árbol
- Árbol binario
- Árbol binario de búsqueda
- Dígrafo
- *Inorden*
- Grafo
- Grafo dirigido
- Grafo no dirigido
- Hoja
- Lista de adyacencia
- Matriz de adyacencia
- Nivel
- Nodo
- *Postorden*
- *Preorden*
- Profundidad
- Raíz
- Rama
- Recorrido de un árbol
- Subárbol

RESUMEN

Las estructuras de datos dinámicas árboles y grafos son muy potentes para la resolución de problemas complejos de tipo gráfico, jerárquico o en red.

La estructura árbol más utilizada normalmente es el **árbol binario**. Un árbol binario es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho.

En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre pero sí cada elemento restante. Cuando un elemento y tiene un padre x, x es un antecesor o antecedente del elemento y.

La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana más 1. Si el árbol A es vacío.

El nivel o profundidad de un elemento es un concepto similar al de altura.

Un árbol binario no vacío está equilibrado totalmente si sus subárboles, izquierdo y derecho, tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados.

Los árboles binarios presentan dos tipos característicos: árboles binarios de búsqueda y árboles binarios de expresiones. Los árboles binarios de búsqueda se utilizan fundamentalmente para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

Los grafos son otra estructura de datos no lineal y que tiene gran número de aplicaciones. Los árboles

binarios representan estructuras jerárquicas con limitaciones de dos subárboles por cada nodo. Si se eliminan las restricciones de que cada nodo puede apuntar a dos nodos —como máximo— y que cada nodo puede estar apuntado por otro nodo —como máximo— nos encontramos con un grafo. *Ejemplos de grafos* en la vida real los tenemos en la red de carreteras de un estado o región, la red de enlaces ferroviarios o aéreos nacionales, etcétera.

Un grafo G consta de dos conjuntos (G = (V, E)): un conjunto V de vértices o nodos y un conjunto E de aristas (parejas de vértices distintos) que conectan los vértices. Si las parejas no están ordenadas, G se denomina *grafo no dirigido*; si los pares están ordenados, entonces G se denomina *grafo dirigido*. El término grafo dirigido se suele también designar como *dígrafo* y el término grafo sin calificación significa grafo no dirigido.

Los grafos se pueden implementar de dos formas típicas: *matriz de adyacencia* y *lista de adyacencia*. La elección depende de las necesidades de la aplicación en concreto, ya que cada una de las formas tiene sus ventajas y sus inconvenientes.

El recorrido de un grafo puede ser en analogía con los árboles, recorrido en profundidad y recorrido en anchura. El recorrido en profundidad es aplicable a los grafos dirigidos y a los no dirigidos y es una generalización del recorrido preorden de un árbol. El *recorrido en anchura* también es aplicable a grafos dirigidos y no dirigidos, que generaliza el concepto de recorrido por niveles de un árbol.

EJERCICIOS

13.1. Dado un árbol binario de números enteros ordenados, se desea un subalgoritmo que busque un elemento con un proceso recursivo.

13.2. Diseñar un subalgoritmo que busque un elemento en un árbol binario de números enteros ordenados, realizado con un proceso repetitivo.

13.3. Describir el orden en el que los vértices de los siguientes árboles binarios serán visitados en: *a*) pre-orden, *b*) in-orden, *c*) post-orden:

13.4. Dibujar la expresión árbol para cada una de las siguientes expresiones y dar el orden de visita a los nodos en: *a*) pre-orden, *b*) in-orden, *c*) post-orden:

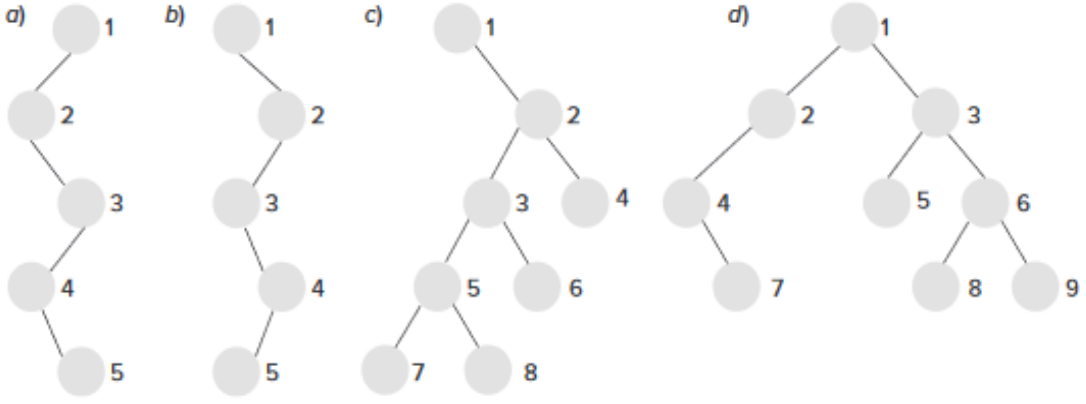
1. log n!
2. $(a - b) - c$
3. $a - (b - c)$
4. $(a < b)$ y $(b < c)$ y $(c < d)$

13.5. Escribir un subalgoritmo recursivo que liste los nodos de un árbol binario en pre-orden.

13.6. Escribir un subalgoritmo que elimine un nodo determinado de un árbol de enteros.

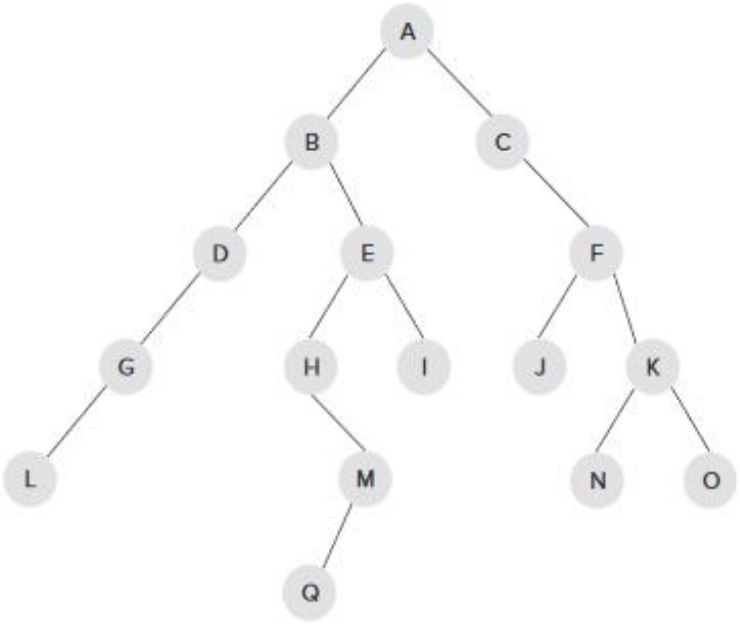
13.7. Se dispone de un árbol de números reales desordenados y se desea escribir un subalgoritmo que inserte un nodo en el lugar correspondiente de acuerdo con su valor.

13.8. Escribir un subalgoritmo que permita conocer el número de nodos de un árbol binario.



Cristian Pilapanta 1095211-2202112342

13.9. Considerar el árbol binario.



Listar los nodos del árbol en: *a*) pre-orden, *b*) in-orden, *c*) post-orden.

Cristian Pilaparita 1095211-2202112342

CAPÍTULO 14

Recursividad

14.1. La naturaleza de la recursividad
14.2. Recursividad directa e indirecta
14.3. Recursión *versus* iteración
14.4. Recursión infinita
14.5. Resolución de problemas complejos con recursividad

CONCEPTOS CLAVE
RESUMEN
EJERCICIOS
PROBLEMAS

Cristian Pilaparita 1095211-2202112342

1095211220211234

INTRODUCCIÓN

La recursividad (recursión) es aquella propiedad que posee una función que puede llamarse a sí misma. Es posible utilizar la recursividad como una alternativa a la iteración. Por lo general, una solución recursiva es menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que llevan consigo las llamadas suplementarias a las funciones; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que en caso contrario serían difíciles de resolver. Por lo tanto, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación.

14.1. La naturaleza de la recursividad[1]

Los programas examinados hasta ahora, generalmente estructurados, se componen de una serie de funciones que llaman unas a otras de modo disciplinado. En algunos problemas es útil disponer de funciones que se llamen a sí mismas. Un *subprograma recursivo* se llama a sí mismo, ya sea directa o indirectamente. La recursividad es un tópico importante examinado frecuentemente en cursos de programación y de introducción a las ciencias de la computación.

En este libro se dará una importancia especial a las ideas conceptuales que soportan la recursividad. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo, numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo.

Hasta el momento, casi siempre se han visto subprogramas que llaman a otros subprogramas distintos. Así, si se dispone de dos procedimientos `proc1` y `proc2`, la organización de un programa tal y como se ha visto hasta este momento podría adoptar una forma similar a ésta:

```
procedimiento proc1(...)
inicio
...
fin_procedimiento

procedimiento proc2(...)
inicio
   ...
   proc1(...)          // llamada a proc1
   ...
fin_procedimiento
```

Al diseñar programas recursivos se tendría esta situación:

```
procedimiento proc1(...)
inicio
   ...
   proc1(...);
   ...
fin_procedimiento
```

o bien esta otra:

```
procedimiento proc1(...)
inicio
   ...
   proc2(...)          //llamada a proc2
   ...
fin_procedimiento

procedimiento proc2(...)
inicio
   ...
   proc1(...)          //llamada a proc1
...
fin_procedimiento
```

[1] La última edición del Diccionario de la Real Academia de la Lengua Española (23a. edición, Madrid, 2014) y actualización de 2019, recogen ya los términos recursividad, recursión (sinónimo de *recursividad*), *recursivo/va* y *recurrente* (www.rae.es).

Cristian Pilaparta 1095211-2202112342

EJEMPLO 14.1

El factorial de un entero no negativo n, escrito n! (y pronunciado n factorial), es el producto:

```
n! = n.(n – 1).(n – 2),...,1
```

en el cual:

```
0! = 1
1! = 1
2! = 2*1 = 2*1!
3! = 3*2*1 = 3*2!
4! = 4*3*2*1 = 4*3!
...
```

así:

```
5! = 5 . 4 . 3 . 2 . 1 = 5 . 4! = 120
```

de modo que una definición recursiva de la función factorial *n* es:

```
n! = n*(n - 1)!     para     n>1
```

El factorial de un entero *n*, mayor o igual a 0, se puede calcular de modo *iterativo* (no recursivo), teniendo presente la definición de *n!* del modo siguiente:

```
n! = 1                  si     n = 0
n! = n * (n – 1)!       si     n > 0
```

El algoritmo que resuelve el factorial de forma iterativa de un entero *n*, mayor o igual que 0, se puede calcular utilizando un bucle `for`:

```
var
entero: contador
  real: factorial
inicio
  ...
  factorial ← 1;
  desde contador ← n hasta 1 decremento 1
     factorial ← factorial * contador
  fin_desde
fin
```

En el caso de implementar una función, se requerirá una sentencia de retorno que devuelva el valor del factorial, tal como:

```
devolver(factorial)
```

El algoritmo que resuelve la función de modo *recursivo* debe contar con una condición de salida. Así, en el caso del cálculo de 6!, la definición es 6! = 6 × 5! y 5! de acuerdo con la definición es 5 × 4! Este proceso continúa hasta que 1! = 1 × 0! por definición. En matemáticas, el método de definición de una función en términos de sí misma se conoce como definición **inductiva** y conduce naturalmente a una implementación recursiva. El caso base de 0! = 1 es esencial dado que se detiene, potencialmente, una cadena de llamadas recursivas. Este caso base o condición de salida deben fijarse en cada caso de una solución recursiva. El algoritmo que resuelve n! de modo recursivo se apoya en la definición siguiente:

```
n! = 1                          si     n = 0
n! = n*(n – 1)*(n – 2)*...*1    si     n > 0
```

Cristian Pilaparia 1095211-2202112342

en consecuencia, el algoritmo mencionado que calcula el factorial será:

```
si (n = 0) entonces
   fac ← 1
si_no
   contador = n - 1
   fac ← n * fac(contador)
fin_si
```

Otro pseudocódigo que resuelve la función factorial es:

```
si n = 1 entonces
   fac ← n
si_no
   fac ← n*fac(n - 1)
fin_si
```

Así, una función recursiva de factorial es:

```
entero función factorial(E entero: n)
   inicio
      si (n = 1) entonces
         devolver (1)
      si_no
         devolver (n * factorial(n - 1))
      fin_si
   fin_función
```

Cristian Pilapanta 1095211-2202112342

Nota

Dado que el valor de un factorial de un número entero aumenta considerablemente a medida que aumenta el valor de *n*, es conveniente en el diseño del algoritmo definir el tipo de dato a devolver por la función como un valor real, al objeto de no tener problema de desbordamiento cuando traduzca a un código fuente en un lenguaje de programación.

EJEMPLO 14.2

Deducir la definición recursiva del producto de números naturales.

El producto *a* * *b*, donde *a* y *b* son enteros positivos, tiene dos soluciones.

Solución iterativa

$$a * b = \underbrace{a + a + a + + \ldots + a}_{b \text{ veces}}$$

Solución recursiva

$$a * b = a \quad \text{si } b = 1$$
$$a * b = a * (b - 1) + a \quad \text{si } b > 1$$

Así, por ejemplo, *7* × *3* será:

$$7 * 3 = 7 * 2 + 7 = 7 * 1 + 7 + 7 = 7 + 7 + 7 = 21$$

En pseudocódigo se tiene:

```
entero función producto (E entero, a, b)
inicio
    si b = 1 entonces
         devolver (a)
    si no
          devolver (a*producto (a, b-1))
    fin_si
fin
```

EJEMPLO 14.3

Definir la naturaleza de la **serie de Fibonacci:** *0, 1, 1, 2, 3, 5, 8, 13, 21, ...*

Se observa en esta serie que comienza con 0 y 1, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores, por ejemplo:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
...
```

Entonces se puede decir que:

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
```

y la definición recursiva será:

```
fibonacci(n) = n      si n = 0   o   n = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)  si n > = 2
```

Obsérvese que la definición recursiva de los números de fibonacci es diferente de las definiciones recursivas del factorial de un número y del producto de dos números. Así, por ejemplo, simplificando el nombre de la función por `fib`:

```
fib(6) = fib(5) + fib(4)
```

o lo que es igual, `fib(6)` ha de aplicarse en modo recursivo dos veces, y así sucesivamente. Las funciones iterativa y recursiva implementadas en Java son:

```
public class Fibonacci
{
   //Fibonacci iterativo
   public static long fibonaccii(int n)
   {
      long f = 0, fsig = 1;
      for (int i = 0; i < n; i++)
      {
         long aux = fsig;
         fsig += f;
         f = aux:
      }
      return(f);
   }
```

Cristian Pilaparta 1095211-2202112342

```
    //Fibonacci recursivo
    public static long fibonaccir(int n)
    {
       // si n es menor que 0 devuelve -1 como señal de error
       if (n < 0)
          return -1;
       // especificar else no es necesario, ya que
       // cuando se ejecuta return se retorna a la sentencia llamadora
       // y la siguiente instrucción ya no se ejecuta
       if (n == 0)
          return(0);
       else
          if (n == 1)
             return(1);
          else
             return(fibonaccir(n-1)+fibonaccir(n-2));
    }
    public static void main(String[] args)
    {
       System.out.println("Fibonacci_iterativo("+8+")="+fibonaccii(8));
       System.out.println("Fibonacci_recursivo("+8+")="+fibonaccii(8));
    }
}
```

Cristian Pilaparita 1095211-2202112342

14.2. Recursividad directa e indirecta

En **recursión directa,** el código del subprograma recursivo F contiene una sentencia que invoca a F, mientras que en **recursión indirecta** el subprograma F invoca al subprograma G que invoca a su vez al subprograma P, y así sucesivamente hasta que se invoca de nuevo al subprograma F.

Si una función, procedimiento o método se invoca a sí misma, el proceso se denomina **recursión directa**; si una función, procedimiento o método puede invocar a una segunda función, procedimiento o método que a su vez invoca a la primera, este proceso se conoce como *recursión indirecta* o *mutua*.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia, la definición recursiva debe incluir un **componente base** (*condición de salida*) en el que *f(n)* se defina directamente (es decir, no recursivamente) para uno o más valores de *n*.

Debe existir una "forma de salir" de la secuencia de llamadas recursivas. Así, en la función *f(n) = n!* para *n* entero:

$$f(n)\begin{cases} 1 & n \le 1 \\ n*f(n-1) & n > 1 \end{cases}$$

la condición de salida o base es $f(n) = 1$ para $n \le 1$.

En el caso de la serie de Fibonacci:

$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$ para $n > 1$.

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo.

C++ permite escribir funciones recursivas. Una función recursiva correcta debe incluir un componente base o condición de salida.

PROBLEMA 14.1

Escribir una función recursiva en C++ que calcule el factorial de un número n y un programa que maneje dicha función.

Recordemos que:

```
n! = 1              si n = 0
n! = n*(n - 1)!     si n ≥ 1
```

La función recursiva que calcula *n!*

```
int Factorial (int n)
{
   // cálculo de n!
   if (n <= 1)
      return 1;
   return n * Factorial(n - 1);
}
```

En el algoritmo anterior se ha considerado que el valor resultante es de tipo entero; sin embargo, observe la secuencia de valores de la función factorial.

n	*n!*
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Como se puede ver, los valores crecen muy rápidamente, y para $n = 8$ ya sobrepasa el valor normal del mayor entero manejado en computadoras de 16 bits (32767). Por consiguiente, será preciso cambiar el tipo de dato devuelto que ha de ser `float`, `double`, `unsigned int`, `long`, etcétera. En consecuencia, el programa `fac.cpp` que calcula el factorial de un número puede ser:

```
//Programa fac.cpp
#include <iostream>
#using namespace std;

// en C++ las funciones han de ser declaradas
// o definidas antes de su uso

double Factorial (int n);
```

Cristian Pilapanta 1095211-2202112343

```
int main()
{
   // declaración
   int núm;
   // escribir ('Por favor introduzca un número: ')
   cout << "Por favor introduzca un número: ";
   // leer(núm)
   cin >> núm;
   // escribir (núm, ' != ', Factorial(núm); endl)
   //endl significa salto de línea
   cout << núm <<" != " << Factorial(núm) << endl;
   // devolver éxito, es decir ejecución válida
   return 0;
}

// definición de la función Factorial
// el paso de parámetros de tipo simple por defecto es por valor
double Factorial (int n)
{
   if (n <=1)
      return (1);
   else
      return (n * Factorial(n - 1));
   // en C++ los paréntesis en la sentencia return son opcionales
}
```

Una variante de este programa podría ser el cálculo del factorial correspondiente a los números naturales 0 a 10. Para ello bastaría sustituir la función `main` anterior por una función tal como ésta e incluir una llamada al archivo `#include <iomanip>`:

```
// Programa principal
int main()
{
   int i;
   for (i = 0; i<=10; i++)
      cout << setw(2) << i <<" != " << Factorial(i) << endl;
      // setw da formato a la salida y establece la anchura del
      // campo a 2
   return 0;
}
```

PROBLEMA 14.2

Escribir una función de Fibonacci de modo recursivo y un programa que manipule dicha función, de modo que calcule el valor del elemento de acuerdo con la posición ocupada en la serie.

Nota

El código fuente de este programa se ha escrito en lenguaje C++.

```
// Función de Fibonacci: fibo.cpp
#include <iostream>
#using namespace std;
```

Cristian Pilaparta 1095211-2202112343

```
long fibonacci (long n);
int main()
{
   long resultado, núm;
   cout << "Introduzca un entero : ";
   cin >> núm;
   resultado = fibonacci (núm);
   cout << "El valor de Fibonacci(" << núm << ")= " << resultado << endl;
   return 0;
}
// definición recursiva de la función de fibonacci
long fibonacci(long n)
{
   if ((n == 0) || (n == 1))
      return n;
   // no es necesaria la especificación de else, pero se puede poner
   return fibonacci(n - 1) + fibonacci (n - 2);
}
```

La salida resultante de la ejecución del programa anterior:

```
Introduzca un entero : 2
El valor de Fibonacci (2) = 1
Introduzca un entero : 20
El valor de Fibonacci (20) = 832040
```

14.2.1. Recursividad indirecta

La recursividad indirecta se produce cuando un subprograma llama a otro, que eventualmente terminará llamando nuevamente al primero. El programa ALFABETO.CPP visualiza el alfabeto utilizando recursión mutua o indirecta.

Nota

El código fuente de este programa también se ha escrito en lenguaje C++.

```
// Listado ALFABETO.CPP
#include <iostream>
#include <stdio.h>
using namespace std;
// A y B equivalen a procedimientos
void A(int c);
void B(int c);
int main()
{
   A('Z');
   cout << endl;
   return 0;
}
void A(int c)
{
   if (c > 'A')
      B(c);
   putchar(c);
}
```

Cristian Pilaparta 1095211-2202112343

```
void B(int c)
{
   A(--c);
}
```

El programa principal llama a la función recursiva `A()` con el argumento `'Z'` (la última letra del alfabeto). La función `A` examina su parámetro `c`. Si `c` está en orden alfabético después que `'A'`, la función llama a `B()`, que inmediatamente llama a `A()`, pasándole un parámetro predecesor de `c`. Esta acción hace que `A()` vuelva a examinar `c`, y nuevamente una llamada a `B()`, hasta que `c` sea igual a `'A'`. En este momento, la recursión termina ejecutando `putchar()` 26 veces y visualizando el alfabeto, carácter a carácter.

14.2.2. Condición de terminación de la recursión

Cuando se implementa un subprograma recursivo será preciso considerar una condición de terminación, ya que en caso contrario el subprograma continuaría indefinidamente llamándose a sí mismo y llegaría un momento en que la memoria se podría agotar. En consecuencia, sería necesario establecer en cualquier subprograma recursivo la condición de parada que termine las llamadas recursivas y evitar indefinidamente las llamadas. Así, por ejemplo, en el caso de la función `factorial`, definida anteriormente, la condición de salida puede ser cuando el número sea 1 o 0, ya que en ambos casos el factorial es 1.

```
real función factorial(E entero: n)
inicio
   si(n = 1) o (n = 0) entonces
      devolver (1)
   si_no
      devolver (n * factorial (n - 1))
   fin_si
fin_función
```

Cristian Pilaparita 1095211-2202112343

1095211220211234

14.3. Recursión *versus* iteración

En las secciones anteriores se han estudiado varias funciones que se pueden implementar fácilmente ya sea de modo recursivo o de modo iterativo. En esta sección compararemos los dos enfoques y examinaremos las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva* y *la recursión utiliza una estructura de selección*. La iteración y la recursión implican repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas; ambas requieren de un test de terminación (*condición de salida*). La iteración termina cuando la condición del bucle no se cumple mientras que la recursión termina cuando se reconoce un caso base o la condición de salida se alcanza.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de recursividad y en consecuencia se necesita tiempo suplementario para realizar las mencionadas llamadas.

Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada de una función recursiva produce que otra copia de la función (realmente sólo las variables de función) sea creada; esto puede consumir memoria considerable. Por el contrario, la iteración se produce dentro de una función, de modo que las operaciones suplementarias de las llamadas a la función y asignación de memoria adicional son omitidas.

En consecuencia, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria, es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

Cualquier problema que se puede resolver recursivamente también puede resolverse de manera iterativa (no recursivamente). Un enfoque recursivo se elige normalmente con preferencia a un iterativo cuando el recursivo es más natural para la resolución del problema y produce un programa más fácil de comprender y depurar. Otra razón para elegir una solución recursiva es que una iterativa puede no ser clara ni evidente.

Consejo de programación

Se ha de evitar utilizar recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional.

Consejo de carácter general

Si la solución de un problema se puede expresar iterativa o recursivamente con igual facilidad, es preferible la solución iterativa, ya que se ejecuta más rápidamente (no existen llamadas adicionales a funciones que consumen tiempo de proceso) y utiliza menos memoria (la pila necesaria para almacenar las sucesivas llamadas necesarias en la recursión). Hay veces, sin embargo, que, pese a todo, es preferible la solución recursiva.

Cristian Pilapaña 1095211-2202112343

EJEMPLO 14.4

La función factorial de un número ya expuesta anteriormente ofrece un ejemplo claro de comparación entre funciones definidas de modo iterativo o modo recursivo y, a continuación, se muestra su implementación en C#.

El factorial *n!*, de un número *n* era:

```
0! = 1
n! = n * (n – 1)! para n > 0
```

Solución recursiva

```
// código en C#
public class Prueba1
{
   // factorial recursivo
   // Precondición    n está definido y n >= 0
   // Postcondición   ninguna
   // Devuelve        n!
   public static long factorial(int n)
   {
      if (n < 0)
         return – 1;
      if (n == 0)
         return 1;
      else
         return n * factorial(n – 1);
   }
  public static void main()
  {
```

```
   // escribir(factorial(4))
   System.Console.WriteLine(factorial(4));
   }
}
```

Solución iterativa

```
// código en C#
public class Prueba2
{
   // factorial iterativo
   // Precondición    n está definido y n >= 0
   // Postcondición   ninguna
   // Devuelve        n!
   public static long factorial(int n)
   {
      if (n < 0)
         return - 1;
      long fact = 1;
      while (n > 0)
      {
         fact = fact * n;
         n = n - 1;
      }
      return fact;
   }
   public static void main()
   {
      // escribir(factorial(4))
      System.Console.WriteLine(factorial(4));
   }
}
```

Cristian Pilapanta 1095211-2202112343

Directrices en la toma de decisión iteración/recursión

1. Considere una solución recursiva sólo cuando una solución iterativa *sencilla* no sea posible.
2. Utilice una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales que se realizan.
4. En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fáciles de leer y comprender que su correspondiente iterativa. En estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el costo extra (en tiempo y memoria) de la ejecución de un programa recursivo.

14.4. Recursión infinita

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida.

En realidad, la **recursión infinita** significa que cada llamada recursiva produce otra y ésta a su vez otra y así para siempre. En la práctica dicho código se ejecutará hasta que la computadora agote la memoria disponible y se produzca una terminación anormal del programa.

El flujo de control de un algoritmo recursivo requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión (*condición de salida* o *caso base*).
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

EJEMPLO 14.5

Se desea calcular la suma de los primeros N enteros positivos.

La función no recursiva que realiza la tarea solicitada es:

```
entero función CalculoSuma (E entero: N)
var
   entero: suma, i
inicio
   suma ← 0
   desde i ← 1 hasta N hacer
      suma ← suma + i
   fin_desde
   devolver (suma)
fin_función
```

La función `CalculoSuma` implementada recursivamente requiere la definición previa de la suma de los primeros `N` enteros matemáticamente en forma recursiva, tal como se muestra a continuación:

$$\text{suma}(N) = \begin{cases} 1 & \textbf{si} \quad N = 1 \\ N + \text{suma}(N-1) & \textbf{\textit{en caso contrario}} \end{cases}$$

La definición anterior significa que si `N` es 1, entonces la función `suma(N)` toma el valor 1; en caso contrario, significa que la función `suma(N)` toma el valor resultante de la suma de `N` y el resultado de `suma(N–1)`. Por ejemplo, la función `suma(5)` se evalúa tal como se muestra en la figura 14.1 de la página siguiente.

El pseudocódigo fuente de la función recursiva `suma` es:

```
entero función suma(E entero: n)
inicio
   // test para parar o continuar (condición de salida)
   si (n = 1) entonces
      devolver (1)
     //caso final – se detiene la recursión
   si_no
      devolver(n + suma (n – 1))
      //caso recursivo
      //la recursión continúa con una llamada recursiva
   fin_si
fin_función
```

Cristian Pilapanta 1095211-2202112343

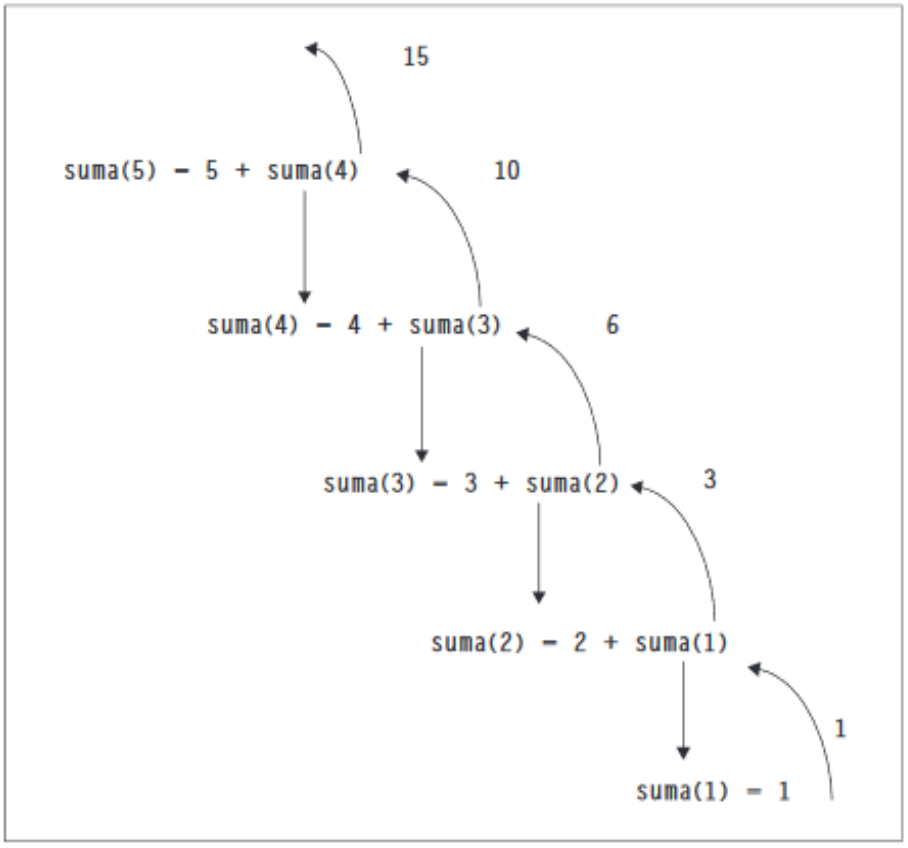


Figura 14.1. Secuencia de llamadas recursivas que evalúan la función `Suma(N)` [en el ejemplo `Suma(4)`].

Cristian Pilapanta 1095211-2202112343

y el código fuente en Turbo Pascal es:

```
program Sumas;
   {solución interactiva}
   function CalculoSuma (N: integer): integer;
   var
      suma, i: integer;
   begin
      suma := 0;
      for i:= 1 to N do
         suma := suma + i;
      CalculoSuma := suma
   end;

   {solución recursiva}
   function suma(n: integer): integer;
   begin
      { test para parar o continuar (condición de salida)}
      if n = 1 then
         suma := 1
      { caso final – se detiene la recursión}
         la recursión continúa con una llamada recursiva }
      else
```

```
        suma := n + suma (n – 1);
   end;
begin
   writeln('Suma recursiva ', suma(4))
   writeln('Suma iterativa ', CalculoSuma(4))
end
```

Cuando se realizan llamadas recursivas se deben pasar argumentos diferentes de los parámetros de entrada; así, en el ejemplo de la función `suma`, el argumento que se pasa en la función recursiva es $n - 1$ y el parámetro es n. La figura 14.2 muestra el flujo de control de la función `suma` de modo recursivo.

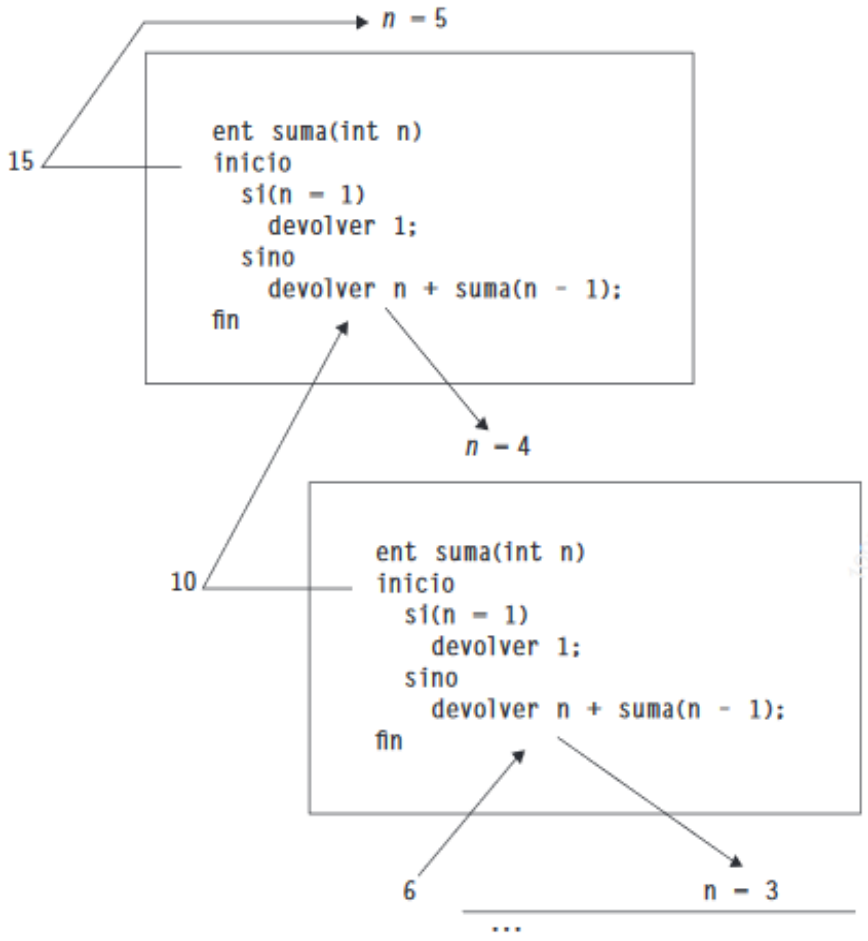


Figura 14.2. Flujo de control de la función suma recursiva.

PROBLEMA 14.3

Deducir cuál es la condición de salida de la función mcd() que calcula el mayor divisor común de dos números enteros b1 y b2 (el **mcd***, máximo común divisor, es el entero mayor que divide a ambos números) y un programa que la manipule.*

El mcd de los enteros *b1* y *b2* se define como el entero mayor que divide a ambos números. El mcd no está definido si *b1* y *b2* son cero. Los valores negativos de *b1* y *b2* se sustituyen por sus valores absolutos. Por ejemplo, supongamos dos números 6 y 124; el procedimiento clásico de obtención del **mcd** es la realización de divisiones sucesivas, se comienza dividiendo ambos números (124 entre 6) si el resto no es 0, se divide el número menor por el resto y así sucesivamente hasta que el resto sea 0.

Cristian Pilapanta 1095211-2202112343

```
124 |6
 04  20

 6 |4        4 |2
 2  1        0  2
```

(*mcd* = 2)

	20	1	2
124	6	4	2
4	2	0	

mcd = 2

En el caso de 124 y 6, el **mcd** es 2. Suponga ahora que los números son *b1=18* y *b2=45:*

```
18 |45
18  0

        45 |18
        09  2

                18 |9
                 0  2
```

(*mcd* = 9)

El **mcd** de 18 y 45 es 9. En consecuencia, la condición de salida es que el resto sea cero. Por tanto:

1. Si *b2* es cero, la solución es *b1*.
2. Si *b2* no es cero, la solución es *mcd(b2, b1 mod b2).*

El código fuente de la función es:

```
entero función mcd(E entero: b1, b2)
inicio
   si (b2 <> 0) entonces    // condición de salida
      devolver ( mcd (b2, b1 mod b2))
   si_no
      devolver (b1)
   fin_si
fin_función
```

Un programa en C++ que gestiona la función `mcd` es `mcd.cpp`.

```
#include <iostream>
using namespace std;

// Programa mcd.cpp, escrito en lenguaje C++
int mcd(int n, int m);

void main()
{
   // datos locales
   int m, n;

   cout << "Introduzca dos enteros positivos :";
```

Cristian Pilapanta 1095211-2202112343

```
   cin >> m >> n;
   cout << endl;
   cout << "El máximo común divisor es : " << mcd(m, n) << endl;
}

// Función recursiva mcd
   int mcd(int n,int m)
   // devuelve el máximo común divisor de m y n
   {
      if (m != 0)               // condición de salida
         return mcd(m, n % m);
      else
         return n;
}   // final de mcd
```

Al ejecutarse el programa se produce la siguiente salida:

```
Introduzca dos enteros positivos : 6   40
El máximo común divisor es : 2
```

El código de la función recursiva en Turbo Pascal es:

```
function mcd (n, m: integer): integer;
begin
   if m <> 0 then
      mcd := mcd(m, n mod m)
   else
      mcd := n
end;
```

Cristian Pilaparita 1095211-2202112343

1095211220211234

14.5. Resolución de problemas complejos con recursividad

Muchos problemas de computadora tienen una formulación simple y elegante que se traduce directamente a código recursivo. En esta sección se describen una serie de ejemplos que incluyen problemas clásicos resueltos mediante recursividad. Entre ellos se destacan problemas matemáticos, las Torres de Hanoi, método de búsqueda binaria, ordenación rápida, árboles de expresión, etcétera. Explicamos con detalle algunos de ellos.

14.5.1. Torres de Hanoi

Este juego (un algoritmo clásico) tiene sus orígenes en la cultura oriental y en una leyenda sobre el Templo de Brahma. El problema en cuestión supone la existencia de 3 varillas o postes en los que se alojaban discos, cada uno es ligeramente inferior en diámetro al que está justo debajo de él, y pretende determinar los movimientos necesarios para trasladar los discos de una varilla a otra cumpliendo las siguientes reglas:

- En cada movimiento sólo puede intervenir un disco.
- Nunca puede quedar un disco sobre otro de menor tamaño.

La figura 14.3 ilustra el problema. Los cuatro discos situados en la varilla I se desean trasladar a la varilla F conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él.

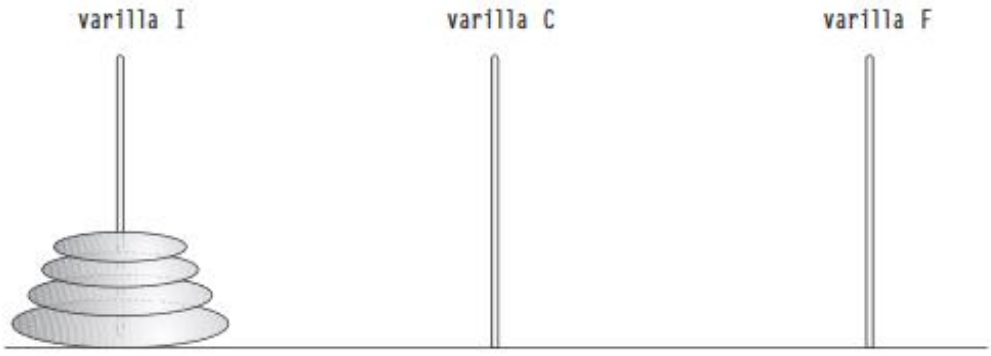


Figura 14.3.

Este problema es claramente recursivo, pues mover cuatro discos de la varilla I a la F consiste en trasladar los tres discos superiores de la varilla origen a otra considerada como auxiliar (C). Observe las figuras 14.3 y 14.4,

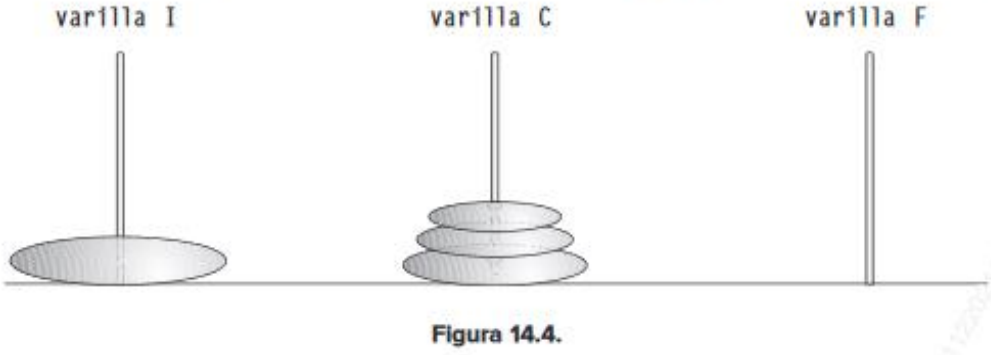


Figura 14.4.

posteriormente, se debe trasladar el disco más grande de la varilla origen al destino (de I a F). Figuras 14.4 (antes) y 14.5 (después),

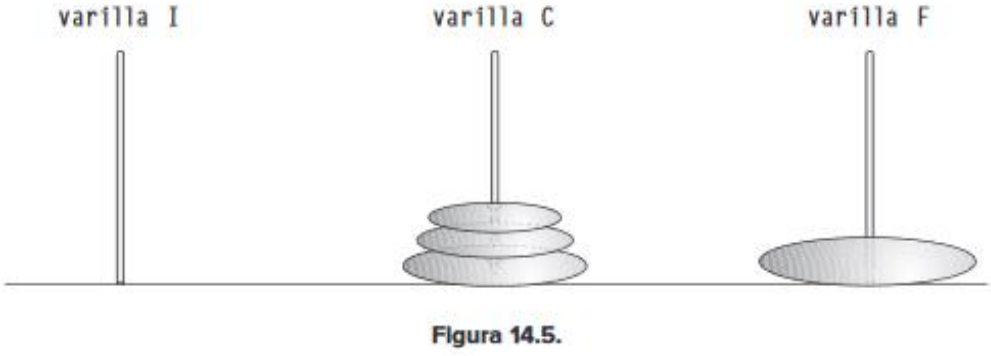


Figura 14.5.

y pasar los tres de la varilla auxiliar al destino, figura 14.5 (antes) y 14.6 (después).

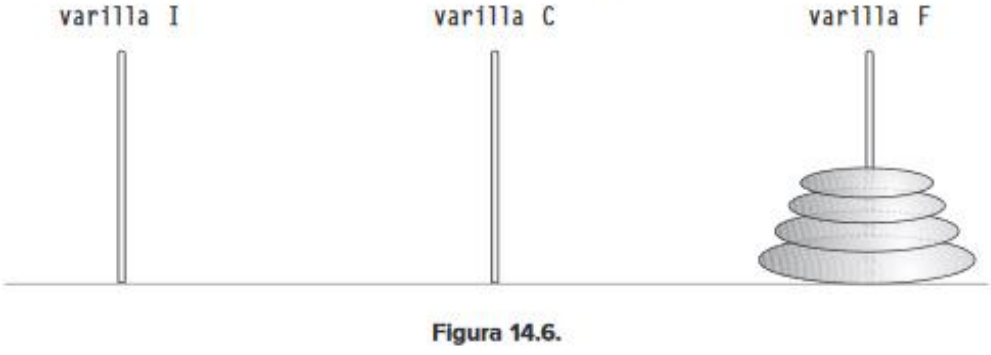
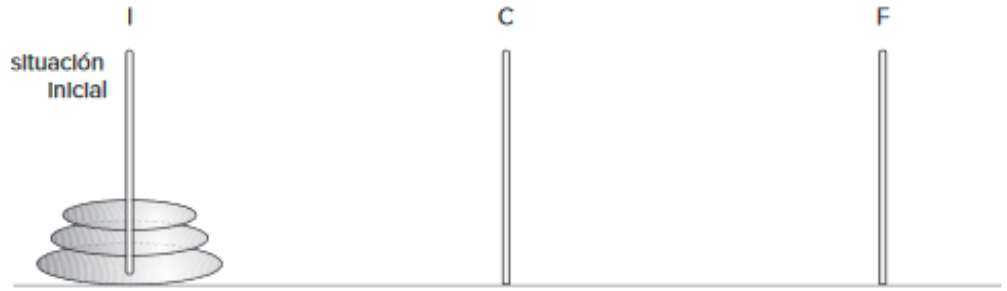


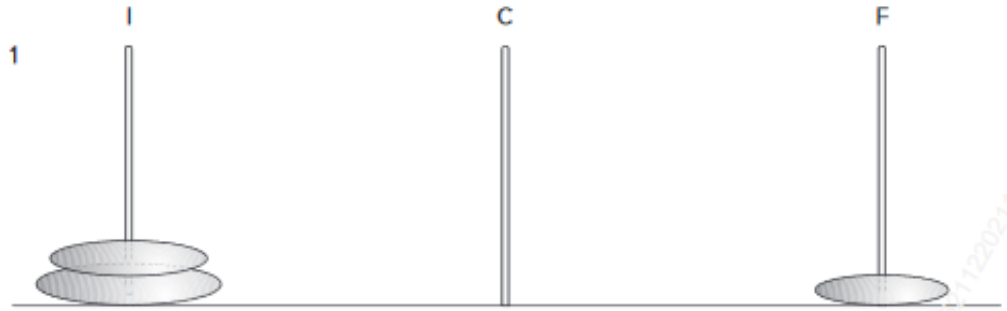
Figura 14.6.

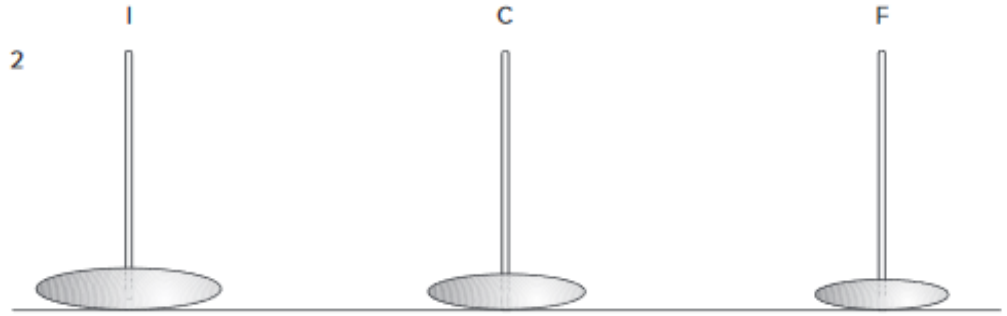
Cristian Pliparán 1095211-2202112343

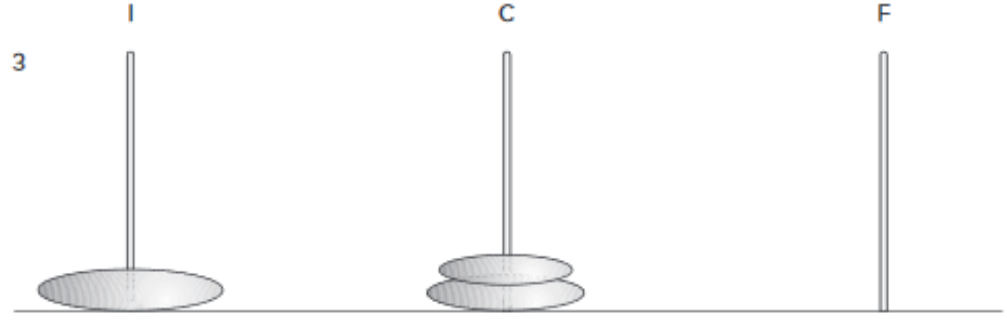
Nuevamente se observa que mover los tres discos superiores de un origen a un destino requiere mover dos de origen a auxiliar, uno de origen a destino y dos de auxiliar a destino. Por último, trasladar dos discos de origen a destino implica trasladar uno de origen a auxiliar, otro de origen a destino y completar la operación pasando el de la varilla auxiliar a destino.

Los movimientos que se realizarían detallados gráficamente para el caso de N = 3, son:

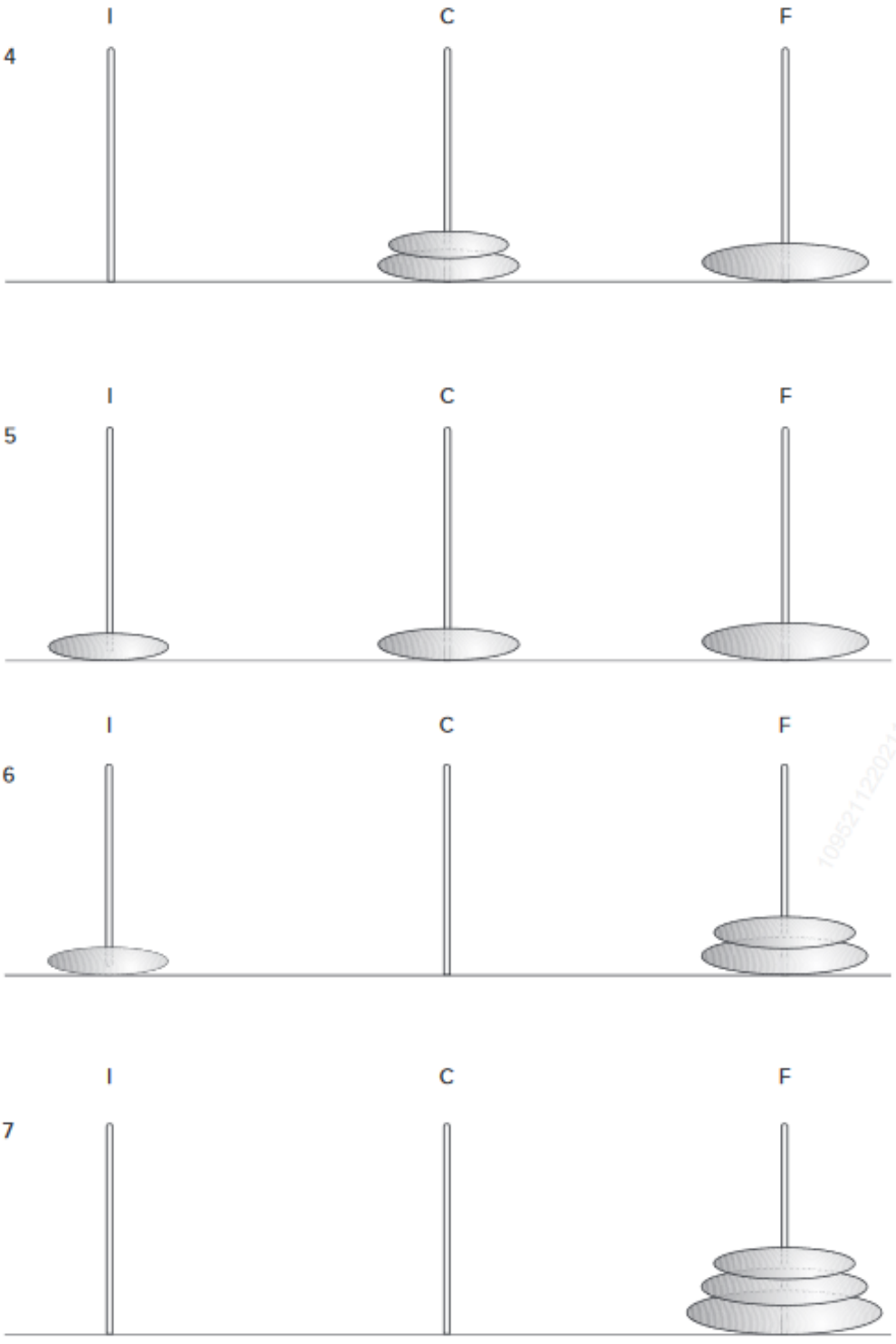








Cristian Pilaparita 1095211-2202112343



Cristian Pilapanta 1095211-2202112344

Diseño del algoritmo

El algoritmo se escribe generalizando para *n* discos y tres varillas. La función de Hanoi declara las varillas o postes como objetos cadena. En la lista de parámetros, el orden de las variables o varillas es:

```
varinicial          varcentral          varfinal
```

lo que implica que se están moviendo discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenar los discos. Si *n = 1* se tiene la condición de parada, ya que se puede manejar moviendo el único disco desde la varilla inicial a la varilla final. El algoritmo sería el siguiente:

```
1. Si n es 1
   1.1 Mover el disco 1 de varinicial a varfinal
2. Si_no
   1.2 Mover n - 1 discos desde varinicial hasta la varilla auxiliar utilizando varfinal
   1.3 Mover el disco n desde varinicial a varfinal
   1.4 Mover n - 1 discos desde la varilla auxiliar o central a varfinal utilizando la vari-
       lla inicial.
```

Es decir, si n es 1, se alcanza la condición de salida o terminación del algoritmo. Si n es mayor que 1, las etapas recursivas 1.2, 1.3 y 1.4 son tres subproblemas más pequeños, que aproximan a la condición de salida.

Las figuras 14.7, 14.8 y 14.9 muestran el algoritmo anterior:

Etapa 1: Mover $n - 1$ discos desde varilla inicial (I).

Figura 14.7.

Etapa 2: Mover un disco desde I a F.

Figura 14.8.

Etapa 3: Mover $n - 1$ discos desde varilla central (C).

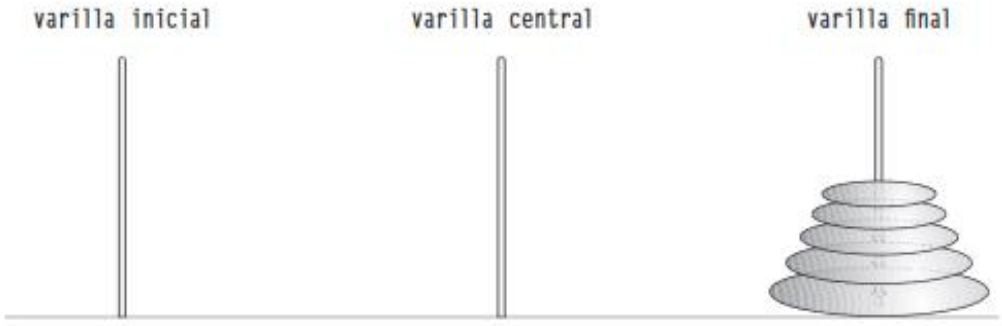


Figura 14.9.

Cristian Pilaparita 1095211-2202112344

La primera etapa en el algoritmo mueve $n - 1$ discos desde la varilla inicial a la varilla central utilizando la varilla final. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es *varinicial*, *varfinal* y *varcentral*.

```
// utilizar varfinal como almacenamiento auxiliar
Hanoi(n – 1, varinicial, varfinal, varcentral);
```

La segunda etapa mueve simplemente el disco mayor desde la varilla inicial a la varilla final:

```
escribir "mover", varinicial, "a", varfinal, fin de línea;
```

La tercera etapa del algoritmo mueve $n - 1$ discos desde la varilla central a la varilla final utilizando `varinicial` para almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es: `varcentral`, `varinicial` y `varfinal`.

```
// utilizar varinicial como almacenamiento auxiliar
Hanoi(n – 1, varcentral, varinicial, varfinal);
```

Implementación de las Torres de Hanoi en C++

La implementación del algoritmo se apoya en los nombres de las tres varillas o alambres `"inicial"`, `"central"` y `"final"` que se pasan como parámetros a la función. El programa comienza solicitando al usuario que introduzca el número de discos *N*. Posteriormente se llama a la función recursiva `Hanoi` para obtener un listado de los movimientos que transferirán los *N* discos desde la varilla `"inicial"` a la varilla `"final"`. El algoritmo requiere $2^N - 1$ movimientos. Para el caso de 10 discos, el juego requerirá 1.023 movimientos; en el caso de prueba para $N = 3$, el número de movimientos es $2^3 - 1 = 7$.

```
// archivo Torres.cpp
// función recursiva Torres de Hanoi

void Hanoi(char varinicial, char varfinal, char varcentral, int n)
{
   if ( n == 1)
      cout << " Mover disco 1 de varilla " << varinicial << " a varilla "
         << varfinal << endl;
   else
   {
      Hanoi(varinicial, varcentral, varfinal, (n – 1);
      cout << " Mover disco " << n << " desde varilla " << varinicial <<
              " a varilla " << varfinal << endl;
      Hanoi(varcentral, varfinal, varinicial, n – 1);
   }
}
```

Una ejecución de la función para el caso de mover tres discos desde las varillas `A` a `C` tomando la varilla `B` como varilla central o auxiliar, se puede conseguir con la siguiente sentencia:

```
Hanoi ('A', 'C', 'B', 3);
```

que resuelve el problema de tres discos desde `A` a `C`. La salida generada sería:

```
Mover   disco  1   de   varilla   A   a   varilla   C
Mover   disco  2   de   varilla   A   a   varilla   B
Mover   disco  1   de   varilla   C   a   varilla   B
Mover   disco  3   de   varilla   A   a   varilla   C
Mover   disco  1   de   varilla   B   a   varilla   A
Mover   disco  2   de   varilla   B   a   varilla   C
Mover   disco  1   de   varilla   A   a   varilla   C
```

Cristian Pilapanta 1095211-2202112344

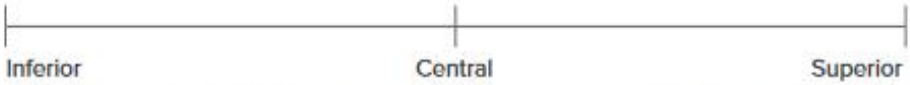
Consideraciones de eficiencia en las Torres de Hanoi

Es de destacar que la función `Hanoi` resolverá el problema de las Torres de Hanoi para cualquier número de discos. El problema de tres discos se resuelve en un total de 7 ($2^3 - 1$) llamadas a la función `Hanoi` mediante 7 movimientos de disco. El problema de cinco discos se resuelve con 31 ($2^5 - 1$) llamadas y 31 movimientos. En general, como ya se ha expresado anteriormente, el número de movimientos requeridos para resolver el problema de n discos es $2^n - 1$. Cada llamada a la función requiere la asignación e inicialización de un área local de datos en la memoria, por lo que el tiempo de computadora se incrementa exponencialmente con el tamaño del problema. Por estas razones, la ejecución del programa con un valor de n mayor que 10 requiere gran cantidad de prudencia para evitar desbordamientos de memoria y ralentización de tiempo.

14.5.2. Búsqueda binaria recursiva

Recordemos que la búsqueda binaria era aquel método de búsqueda de una clave especificada dentro de una lista o array ordenado de n elementos que realizaba una exploración de la lista hasta que se encontraba o no la coincidencia con la clave especificada. El algoritmo de búsqueda binaria se puede describir recursivamente.

Suponga que se tiene una lista ordenada `A` con un límite inferior y un límite superior. Dada una clave (valor buscado) se comienza la búsqueda en la posición central de la lista (índice central).

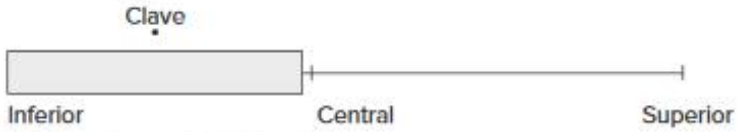



```
Central = (inferior + superior)div 2
```

Comparar `A[central]` y `clave`

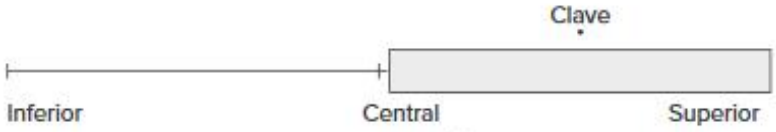
Si se produce coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. Si no se produce la coincidencia (no se encuentra la clave), dado que la lista está ordenada, se centra la búsqueda en la "sublista inferior" (a la izquierda de la posición central) o en la "sublista derecha" (a la derecha de la posición central).

1. Si `clave < A[central]`, el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango `inferior` a `central - 1`.



Búsqueda en sublista izquierda
[inferior..central –1]

2. Si `clave > A[central]`, el valor buscado sólo puede entrar en la mitad derecha de la lista con elementos en el rango de índices, `Central + 1` a `Superior`.



Búsqueda en sublista derecha
[central + 1..superior]

3. El proceso recursivo continúa la búsqueda en sublistas más y más pequeñas. La búsqueda termina con éxito (*aparece la clave buscada*) o sin éxito (*no aparece la clave buscada*), situación que ocurrirá cuando el límite superior de la lista sea más pequeño que el límite inferior. La condición `Inferior > Superior` será la condición de salida o terminación y el algoritmo devuelve el índice – 1.

Cristian Pilapaña 1095211-2202112344

En notación matemática y algorítmica se podría representar la búsqueda binaria de la siguiente forma:

```
BusquedaBR(inferior, superior, clave)   // BR, binaria recursiva devolver no encontrada

    devolver no encontrada
       si inferior > superior
    devolver central
       si elemento[central] = clave
=   devolver BusquedaBR(central + 1, superior, clave)
       si elemento[central] < clave
    devolver BusquedaBR(inferior, central - 1, clave
       si elemento[central] > clave
```

en donde `central` es el punto central entre `inferior` y `superior`. Su codificación en Java podría ser:

```
public class Bbin
{
   private int busquedaBinaria(int[] a, int iz, int de, int c)
   {
      int central;

      if (de < iz)
         return(- 1);
      else
      {
         central = (iz + de)/2;
         if (c < a[central])
            return(busquedaBinaria(a, iz, central - 1, c));)
         else
            if (a[central] < c)
               return(busquedaBinaria(a, central + 1, de, c));
            else
               return(central);
   }
  }

   public int busquedaB(int[] a, int c)
   {
      // los arrays en Java comienzan con el subíndice 0
      // a.length se encuentra predefinido y devuelve
      // la longitud del array
      return(busquedaBinaria(a,0,a.length - 1,c));
   }
}
```

14.5.3. Ordenación rápida (*quicksort*)

El algoritmo conocido como *quicksort* (ordenación rápida) recibe el nombre de su autor, Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar. El método es, posiblemente, el más pequeño de código, más rápido, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación. El método consiste en dividir los *n* elementos de la lista a ordenar en tres partes o particiones: una partición *izquierda*, una *central* que sólo contiene un elemento denominado *pivote* o elemento de partición y una partición *derecha*. La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

Cristian Pilaparia 1095211-2202112344

La lista se divide en particiones (sublistas) eligiendo uno de los elementos de la lista y se utiliza como *pivote* o *elemento de partición*. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede elegir cualquier elemento de la lista como pivote; por ejemplo, el primer elemento. Si la lista tiene algún orden parcial, que se conoce, se puede tomar otra decisión para el pivote. Idealmente, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad, de acuerdo con el tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones "pobres" de pivotes.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y la otra en la que todos los elementos (claves) son mayores que o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o "particionado" recursivo de la lista hasta que todas las sublistas constan de sólo un elemento.

EJEMPLO 14.6

1. *Lista inicial* 2 96 18 38 12 45 10 55 81 43 39

   pivote elegido 39

2	10	18	38	12	39	96	55	81	43	45

   <= pivote (2 10 18 38 12) — pivote (39) — >= pivote (96 55 81 43 45)

2. *Lista inicial* 13 81 92 43 65 31 57 26 75 0

   pivote

   0 13 92 43 65 31 57 26 75 81

   <= pivote (0) — pivote (13) — >= pivote (92 43 65 31 57 26 75 81)

EJEMPLO 14.7

1. *Lista original* 5 2 1 8 3 7 9

   pivote *elegido* 5

   sublista izquierda1, Izqda1 (elementos menores que 5) 2 1 3

   sublista derecha1, Dcha1 (elementos mayores o iguales a 5) 8 7 9

2. *Sublista Izda1* 2 1 3

   pivote2

   sublista Izda2 1

   sublista Dcha2 3

   Sublista Izda1 Izda pivote2 Dcha

   1 2 3

3. *Sublista Dcha1* 8 7 9

   pivote3

   sublista Izda2 7

   sublista Dcha2 9

Cristian Pilaparita 1095211-2202112344

Sublista Dcha1

Izda	pivote3	Dcha
7	8	9

4. *Lista ordenada final*

Sublista izquierda			*Pivote*	*Sublista derecha*		
1	2	3	5	7	8	9

El algoritmo *quicksort* requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido, aunque la estrategia de partición suele ser similar. La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se debe buscar el sistema para situar en la sublista izquierda todos los elementos menores o iguales que el pivote y en la sublista derecha todos los elementos mayores que el pivote y dejar el pivote como separador de ambas sublistas.

EJEMPLO 14.8

Lista: 8 1 4 9 6 3 5 2 7 0

Etapa 1:

En esta etapa se efectúa la selección del pivote. Lo primero que se hace es calcular la posición central y si el primer elemento es mayor que el central se intercambian,

Lista: 6 1 4 9 8 3 5 2 7 0

si el primer elemento es mayor que el último, se intercambian,

Lista: 0 1 4 9 8 3 5 2 7 6

si el central es mayor que el último, se intercambian.

Lista: 0 1 4 9 6 3 5 2 7 8

Se toma ahora el central como pivote y se intercambia con el elemento extremo.

Pivote 6
Lista: 0 1 4 9 8 3 5 2 7 6

Etapa 2:

La etapa 2 requiere mover todos los elementos menores al pivote, entre el primero y el penúltimo, a la parte izquierda del array y los elementos mayores a la parte derecha.

0 1 4 9 8 3 5 2 7 6

Para ello se recorre la lista de izquierda a derecha utilizando un contador `i` que se inicializa en la posición más baja (`Inferior`) buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un contador `j` inicializado en la posición más alta, `Superior-1`.

El contador `i` se detiene en el elemento 9 (mayor que el pivote) y el contador `j` se detiene en el elemento 2 (menor que el pivote).

Ahora se intercambian 9 y 2 para que estos dos elementos se sitúen correctamente en cada sublista.

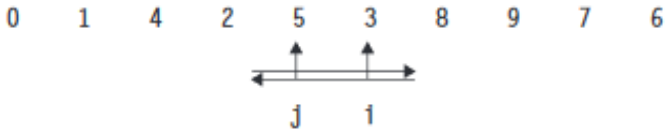
0 1 4 2 8 3 5 9 7 6

Cristian Pilquimán 19952112-2002112344

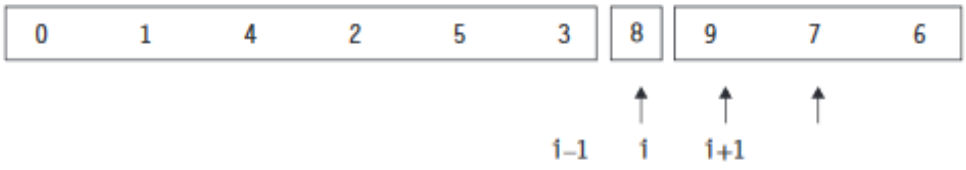
A medida que el algoritmo continúa, i se detiene en el elemento mayor, 8, y j se detiene en el menor, 5.

Se intercambian los elementos mientras que i y j no se cruzan, por tanto se intercambian 8 y 5.

Continúa la exploración.



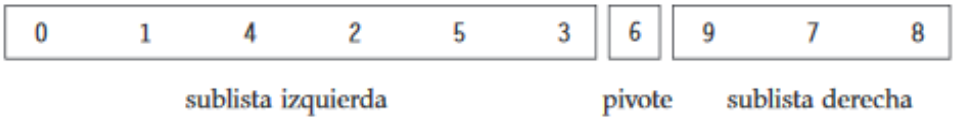
En esta posición los contadores i y j se encuentran sobre el mismo elemento del array y en este caso se detiene la búsqueda y no se realiza ningún intercambio, ya que el elemento al que accede el contador j está ya correctamente situado. Las dos sublistas ya han sido creadas (la lista original se ha dividido en dos particiones).



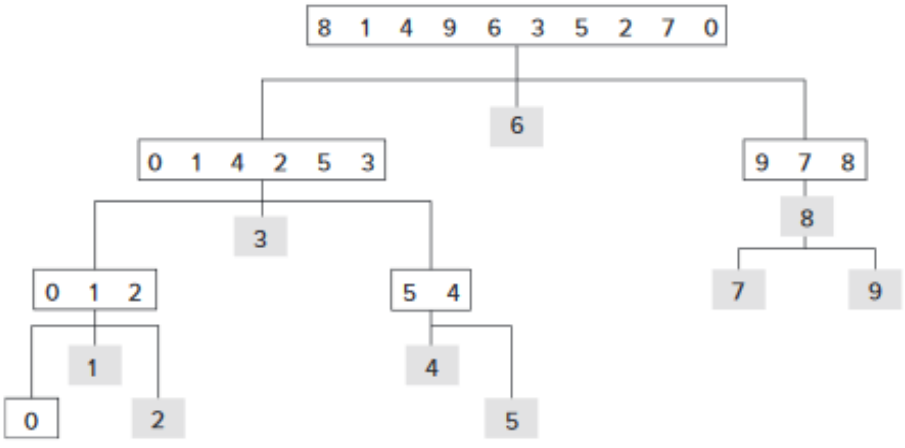
Ahora lo único que se necesita es intercambiar el elemento que está en la posición i con el elemento pivote.

Etapa 3:

Intercambiar el elemento de la posición i con el pivote, de modo que se tendrá la secuencia prevista inicialmente:



Resumiendo el proceso general sería:



Cristian Pilapanta 1095211-2202112344

14.5.3.1. Algoritmo *quicksort*

El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el pivote. Aunque la posición del pivote en principio puede ser cualquiera, una de las decisiones más ponderadas es aquella que lo considera el elemento central o próximo al central de la lista. La figura 14.10 muestra las operaciones del algoritmo para ordenar la lista de elementos enteros L.

```
// algoritmo quicksort
// ordenar a[0:n-1]

Seleccionar un elemento de a[0:n-1] como elemento central
     (este elemento es el pivote)
Dividir los elementos restantes en particiones izquierda y derecha,
   de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que el pivote
   y que ningún elemento a la derecha tenga una clave más pequeña que la del pivote.
Ordenar la partición izquierda utilizando quicksort recursivamente.
Ordenar la partición derecha utilizando quicksort recursivamente.
```

14.5.4. Ordenación *mergesort*

La idea básica de este método de ordenación es la mezcla (*merge*) de listas ya ordenadas. El algoritmo puede considerarse que aplica la técnica "divide y vence", el proceso es simple: si se ordena la primera mitad de la lista, se ordena la segunda mitad y, una vez ordenadas, se mezclan, dando lugar a una lista de elementos ordenada. A su vez, la ordenación de la sublista mitad sigue los mismos pasos, ordenar la primera mitad, ordenar la segunda mitad y mezclar. La sucesiva división de la lista actual en dos hace que el problema (número de elementos) cada vez sea más pequeño; así hasta que la lista actual tenga un elemento y, por tanto, se considera ordenada, es el caso base y a partir de dos sublistas de un número mínimo de elementos se mezclan, dando lugar a listas ordenadas de cada vez más elementos hasta alcanzar la lista total.

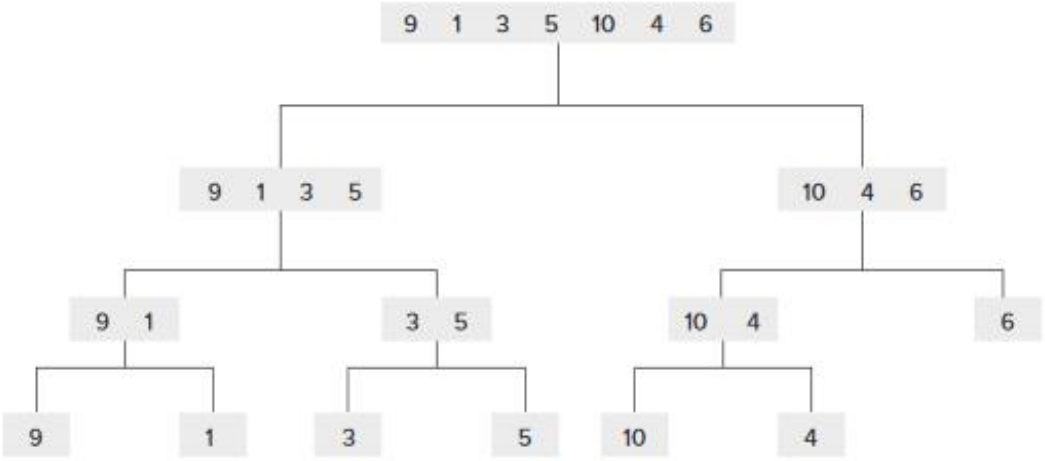
Es decir, que el método consiste, pues, en dividir el vector por su posición central en dos partes y tratar análogamente cada una de ellas hasta que consten de un único elemento. Hay que tener en cuenta que un vector con un único elemento siempre se encuentra ordenado. A la salida de los procesos recursivos las partes ordenadas (subvectores) se mezclan de forma que resultan otras, de mayor longitud, también ordenadas.

EJEMPLO 14.9

Seguir la estrategia del algoritmo mergesort *para ordenar la lista:*

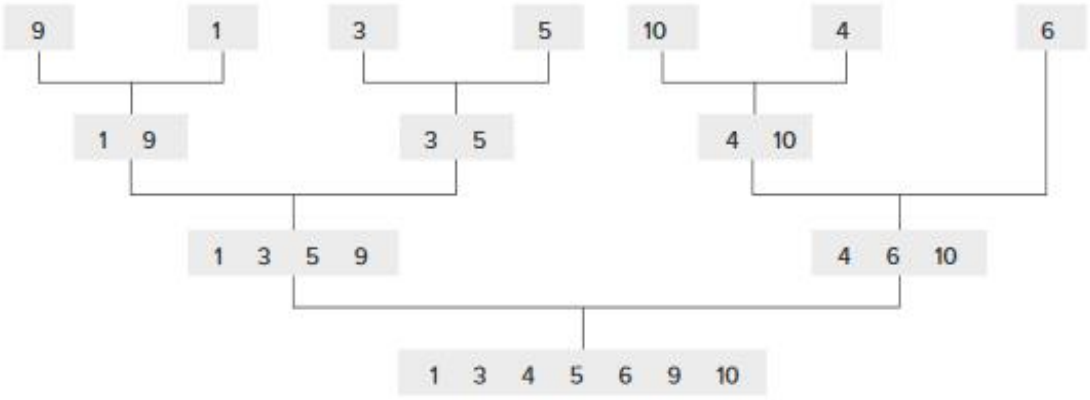
9 1 3 5 10 4 6

Se representa el proceso con las siguientes figuras en las que aparecen las divisiones.



Cristian Pilaparia 1095211-2202112344

La mezcla comienza con las sublistas de un solo elemento, que dan lugar a otra sublista del doble de elementos ordenados. El proceso continúa hasta que se construye una única lista ordenada. A continuación se muestra la creación de las sublistas ordenadas:



14.5.4.1. Algoritmo *mergesort* en Java

Este algoritmo de ordenación se diseña fácilmente con ayuda de las llamadas recursivas para dividir las listas en dos mitades; posteriormente se invoca al método de mezcla de dos listas ordenadas. La delimitación de las dos listas se puede hacer con tres índices: `primero`, `central` y `último`, que apuntan a los elementos del array significados por los identificadores. Así, si se tiene una lista de 10 elementos los valores de los índices:

```
primero = 0; último = 9; central = (primero+último)/2 = 4
```

La primera sublista comprende los elementos a_0 .. a_4 y la segunda los elementos siguientes a_{4+1} .. a_9. Los pasos del algoritmo *mergesort* para el *array* (arreglo) a:

```
procedimiento mergesort(E/S arr: a, E entero: primero, último)
   inicio
      Si primero < último entonces
         central ← (primero+último) div 2
         mergesort(a, primero, central)
         // ordena primera mitad de la lista
         mergesort(a, central+1, último)
         // ordena segunda mitad de la lista
         mezcla(a, primero, central, último)
         {fusiona las dos sublistas ordenadas, delimitadas por
         los extremos}
   fin_si
fin_procedimiento
```

La *codificación* en **Java** consta del método `mergesort()` y del método auxiliar `mezcla()`.

```
public class PruebaMS
{
   public void mergesort(double[] a, int primero, int último)
   {
      int central;
      if (primero < último)
      {
         central = (primero+último)/2;
         // división entera puesto que los operandos son enteros
```

Cristian Pilaparia 1095211-2202112344

```
            mergesort(a, primero, central);
            mergesort(a, central+1, último);
            mezcla(a, primero, central, último);
        }
    }
    private void mezcla(double[] a, int izda, int medio, int drcha)
    {
        double [] tmp = new double[a.length];
        int x, y, z;
        x = z = izda;
        y = medio+1;
            // bucle para la mezcla, utiliza tmp[] como array auxiliar
        while (x<=medio && y<=drcha)
        {
            if (a[x] <= a[y])
                tmp[z++] = a[x++];
            else
                tmp[z++] = a[y++];
        }
        // bucle para mover elementos que quedan de sublistas
        while (x <= medio)
            tmp[z++] = a[x++];
        while (y <= drcha)
            tmp[z++] = a[y++];

        // Copia de elementos de tmp[] al array a[]

        System.arraycopy(tmp, izda, a, izda, drcha-izda+1);
    }
    public static void main(String[] args)
    {
        PruebaMS up=new PruebaMS();
        double[] a = {9,1,3,5,10,4,6};
        up.mergesort(a, 0, a.length-1);
        for (int i = 0; i < a.length; i++)
            System.out.println(a[i]);
    }
}
```

CONCEPTOS CLAVE

- Clases de recursividad: directa e indirecta
- Concepto de recursividad
- Complejidad de los métodos de ordenación
- Eficiencia en cuanto al tiempo de ejecución
- Iteración *versus* recursión
- *Notación O*
- Requisitos de un algoritmo recursivo

RESUMEN

Un subprograma se dice que es recursivo si tiene una o más sentencias que son llamadas a sí mismo. La recursividad puede ser directa e indirecta, la última ocurre cuando el subprograma —método, procedimiento o función— `f()` llama a `p()` y éste a su vez llama a `f()`. La recursividad es una alternativa a la

Cristian Pilapanta 1095211-2202112344
1095211220211234

iteración en la resolución de algunos problemas matemáticos. Los aspectos más importantes a tener en cuenta en el diseño y construcción de métodos recursivos son los siguientes:

- Un algoritmo recursivo correspondiente con un método normalmente contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación o parada del problema en los que éste se soluciona sin ninguna llamada recursiva sino con una sentencia simple. De otro modo, un método recursivo debe tener dos partes: una de terminación en la que se deja de hacer llamadas, es el caso base, y una llamada recursiva con sus propios parámetros.
- Muchos problemas tienen naturaleza recursiva y la solución más fácil es mediante un método recursivo. De igual modo, aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante algoritmos iterativos.
- Todo algoritmo recursivo puede ser transformado en otro de tipo iterativo, pero para ello a veces se necesita utilizar pilas donde almacenar los cálculos parciales.
- Los métodos con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite, que depende de la memoria de la computadora; en caso de superar este límite ocurre un error de overflow.
- Cuando se codifica un método recursivo se debe comprobar siempre que tiene una condición de terminación; es decir, que no se producirá una recursión infinita. Durante el aprendizaje de la recursividad es usual que se produzca ese error.
- Para asegurarse de que el diseño de un método recursivo es correcto se deben cumplir las siguientes tres condiciones:

1. No existe recursión infinita. Una llamada recursiva puede conducir a otra y ésta conducir a otra, y así sucesivamente; pero cada llamada debe de aproximarse más a la condición de terminación.
2. Para la condición de terminación, el método devuelve el valor correcto para ese caso.
3. En los casos que implican llamadas recursivas: si cada uno de los métodos devuelve un valor correcto, entonces el valor final devuelto por el método es el valor correcto.

Cristian Pilapanta 1095211-2202112344

EJERCICIOS

14.1. La suma de una serie de números consecutivos de 1 se puede definir recursivamente como:

```
suma(1) = 1
suma(n) = n + suma(n-1)
```

Escribir la función recursiva que acepte *n* como un argumento y calcule la suma de los números de 1 a *n*.

14.2. El valor de x^n se puede definir recursivamente como:

$x^0 = 1$

$x^n = x * x^{n-1}$

Escribir una función recursiva que calcule y devuelva el valor de x^n.

14.3. Reescribir la función escrita en el ejercicio 14.2 de modo que se utilice un algoritmo repetitivo para calcular el valor de x^n.

14.4. Convierta la siguiente función iterativa en una recursiva. La función calcula un valor aproximado de *e*, la base de los logaritmos naturales, sumando las series

$1 + 1/1! + 1/2! + \ldots + 1/n!$

hasta que los términos adicionales no afecten a la aproximación

```
real función loge()
   var
   // Datos locales
   real: enl, delta, fact
   entero: n
inicio
   enl ← 1.0
   fact ← 1.0
   delta ← 1.0
   hacer
      enl ← delta
      n ← n + 1
      fact ← fact * n
      delta ← 1.0 / fact
```

```
   mientras (enl <> enl + delta)
   devolver (enl)
fin_función
```

14.5. Explique por qué la siguiente función puede producir un valor incorrecto cuando se ejecute:

```
real función factorial (E real: n)
inicio
   si (n = 0 o n = 1) entonces
      devolver(1)
   si_no
      devolver (n * factorial (n-1))
   fin_si
fin_función
```

14.6. Proporcionar funciones recursivas que representen los siguientes conceptos:

a) El producto de dos números naturales.
b) El conjunto de permutaciones de una lista de números.

14.7. El elemento mayor de un array entero de *n-elementos* se puede calcular recursivamente. Definir la función:

```
entero función max(E entero: x, y)
```

que devuelve el mayor de dos enteros *x* y *y*. Definir la función

```
entero función maxarray(E arr: a,
E entero: n)
```

que utiliza recursión para devolver el elemento mayor de *a*:

Condición de parada: `n == 1`

Incremento recursivo: `maxarray = max(max(a[0]... a[n–2]),a[n–1])`

PROBLEMAS

14.1. *La expresión matemática C(m, n) en el mundo de la teoría combinatoria de los números representa el número de combinaciones de m elementos tomados de n en n elementos*

$$C(m, n) = \frac{m!}{n!(m-n)!}$$

Escribir y probar una función que calcule *C(m, n)* donde *n!* es el factorial de *n*.

14.2. *Un palíndromo es una palabra que se escribe exactamente igual leído en un sentido o en otro. Palabras tales como level, deed, ala, etcétera, son ejemplos de palíndromos. Escribir una función recursiva que devuelva un valor de 1 (verdadero), si una palabra pasada como argumento es un palíndromo y devuelva 0 (falso) en caso contrario.*

14.3. *La suma de los primeros n números enteros responde a la fórmula:*

$$1 + 2 + 3 + \ldots + n = n(n+1)/2$$

Inicializar el array `A` que contiene los primeros 50 enteros. La media de estos elementos del array es entonces 51/2 = 25.5. Comprobar la solución aplicando la función recursiva `media (media (float a[], int n))`.

14.4. Leer un número entero positivo $n < 10$. Calcular el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^2 en la forma x^{**1}.

Sugerencia:

$$(x+1)^n = C_{n,n}x^n + C_{n,n-1}x^{n-1} + C_{n,n-2}x^{n-2} + \ldots + C_{n,2}x^2 + C_{n-1}x^1 + C_{n10}x^0$$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de *n*.

La relación de recurrencia de los coeficientes binomiales es:

$$C(n, 0) = 1$$
$$C(n, n) = 1$$
$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

Estos coeficientes constituyen el famoso Triángulo de Pascal y será preciso definir la función que genera el triángulo

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
...
```

14.5. *Escribir un programa en el que el usuario introduzca 10 enteros positivos y calcule e imprima su factorial.*

Cristian Pilaparita 1095211-2202112344

PARTE III

PROGRAMACIÓN ORIENTADA A OBJETOS Y UML 2.5.1

Cristian Pilapanta 1095211-2202112344

CONTENIDO

CAPÍTULO 15. Tipos abstractos de datos, objetos y modelado con UML 2.5.1

CAPÍTULO 16. Diseño de clases y objetos: representaciones gráficas en UML

CAPÍTULO 17. Relaciones entre clases: delegaciones, asociaciones, agregaciones, herencia

CAPÍTULO 15

Tipos abstractos de datos, objetos y modelado con UML 2.5.1

15.1. Programación estructurada (*procedimental*)
15.2. Programación orientada a objetos
15.3. Modelado e identificación de objetos
15.4. Propiedades fundamentales de orientación a objetos
15.5. Modelado de aplicaciones: UML
15.6. Modelado y modelos
15.7. Diagramas de UML 2.5.1
15.8. Bloques de construcción (componentes) de UML 2.5.1
15.9. Especificaciones de UML
15.10. Historia de UML
CONCEPTOS CLAVE
RESUMEN
ANEXO
EJERCICIOS

Cristian Pilapanta 1095211-2202112344

1095211220211234

INTRODUCCIÓN

La *Programación orientada a objetos* (**POO**) es un enfoque conceptual específico para diseñar programas, utilizando un lenguaje de programación orientado a objetos, en nuestro caso C++ o Java. Las propiedades más importantes de la POO son:

- Abstracción.
- Encapsulamiento y ocultación de datos.
- Polimorfismo.
- Herencia.
- Reusabilidad o reutilización de código.

Este paradigma de programación supera las limitaciones que soporta la programación tradicional o "procedimental" y por esta razón comenzaremos el capítulo con una breve revisión de los conceptos fundamentales de este paradigma de programación.

Los elementos fundamentales de la POO son las clases y objetos. En esencia, la POO se concentra en el objeto tal como lo percibe el usuario, pensando en los datos que se necesitan para describirlo y las operaciones que representarán la iteración del usuario con los datos. Después se desarrolla una descripción de la interfaz externa y se decide cómo implementar la interfaz y el almacenamiento de datos. Por último, se ponen juntos en un programa que utilice su nuevo dueño.

En este texto nos limitaremos al campo de la programación, pero también es posible hablar de sistemas de administración de bases de datos orientadas a objetos, sistemas operativos orientados a objetos, interfaces de usuarios orientadas a objetos, etcétera.

En el capítulo se hace una introducción a **UML**, como **Lenguaje Unificado de Modelado**. UML se ha convertido *de facto* en el estándar para modelado de aplicaciones software y es un lenguaje con sintaxis y semántica propias, se compone de pseudocódigo, código real, programas, etcétera. Además, en este capítulo se presentará una breve historia de UML desde la *ya* mítica versión 0.8 hasta *la actual versión 2.5.1* cuyas especificaciones fueron aprobadas en diciembre de 2017.

15.1. Programación estructurada (*procedimental*)

La programación estructurada que se ha estudiado en profundidad hasta este momento viene representada por *lenguajes procedimentales* clásicos como C y Pascal, aunque los más antiguos como Fortran, Cobol o Basic también pertenecen a esta categoría. En estos lenguajes, cada sentencia (instrucción) del lenguaje indica a la computadora que debe realizar alguna acción o tarea. "Obtener una entrada", "sumar dos números", "dividir por cinco", "visualizar la salida", entre otras. En un lenguaje procedimental, un programa es una lista (conjunto) de instrucciones o sentencias.

En el caso de pequeños programas, no se necesita ningún otro principio de organización. El programador crea una lista de instrucciones y una computadora las ejecuta. Cuando los programas se vuelven más complejos, la lista de instrucciones se vuelve grande e inmanejable ya que fácilmente se alcanzan decenas o centenas de instrucciones. En este caso el programa se rompe en unidades más pequeñas que se hagan más comprensibles a las personas que lo utilizan. Estas unidades en C (precursor de C++) se denominan *funciones* (término utilizado en C/C++/Java/Python en otros lenguajes el mismo concepto se conoce por el término de *subrutina, procedimiento* o *subprograma*). Un programa procedimental se divide en funciones (idealmente, al menos, cada función tiene un propósito claramente bien definido y una interfaz también bien definida a las otras funciones del programa).

La idea de romper un programa en funciones se extiende al agrupamiento de un número determinado de funciones en una entidad más grande llamada **módulo** (que normalmente se agrupa en un **archivo** o **fichero**). El principio siempre es el mismo: agrupar componentes que ejecutan una lista de instrucciones. La división de un programa en funciones y módulos es una de las características fundamentales de la **programación estructurada** y que facilita la lectura y comprensión del programa.

Desde un punto de vista de conceptos prácticos de programación, los lenguajes de computadoras tratan dos conceptos fundamentales: datos y algoritmos. Los **datos** constituyen la información que utiliza y procesa un programa. Los **algoritmos** son los métodos que utiliza el programa (instrucciones paso a paso que conducen a la solución del programa). La ecuación fundamental de la programación estructurada, debida a Niklaus Wirth es:

$$Algoritmos + Datos = Programas$$

La programación estructurada utiliza fundamentalmente instrucciones **secuenciales**, de **selección** (`if-then`, `case`) y **repetitivas** (`for`, `while` y `do-while`) para facilitar la realización de tareas secuenciales, selectivas o de decisión y repetitivas o iterativas. El diseño del programa estructurado se consigue rompiendo un programa (o un problema) grande en unidades o tareas más pequeñas y manejables llamadas **funciones** (en C, C++, Python o Java).

15.1.1. Limitaciones de la programación estructurada

Cuando el problema a resolver es complicado, la programación se hace difícil y excesivamente compleja. Las dificultades provienen de que las funciones tienen acceso ilimitado a datos globales y además el paradigma o enfoque procedimental proporcionan un modelo pobre del mundo real.

Desde el punto de vista de un lenguaje procedimental, como C, existen dos tipos de datos: locales y globales. Los **datos locales** están ocultos en el interior de la función y se utilizan exclusivamente por la función. Los **datos globales** son aquellos que pueden ser accedidos por cualquier función del programa.

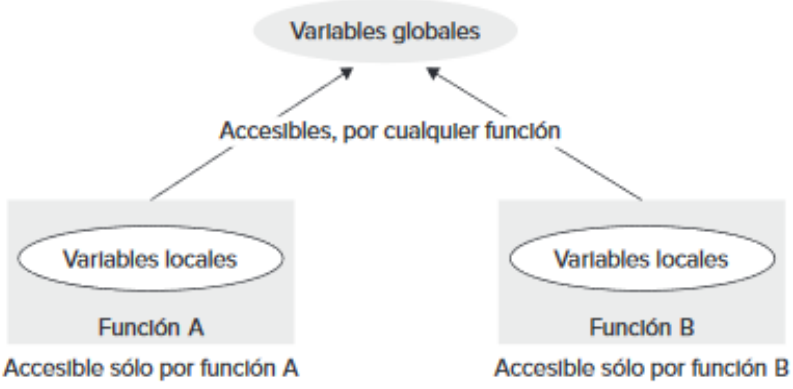


Figura 15.1. Datos locales y globales.

Cristian Pilapanta 1095211-2202112344

En un programa grande existen muchas funciones y muchos datos globales y eso conduce a un número muy grande de posibles conexiones entre ellos. Lo que dificulta la conceptualización de la estructura del programa y la modificación del propio programa.

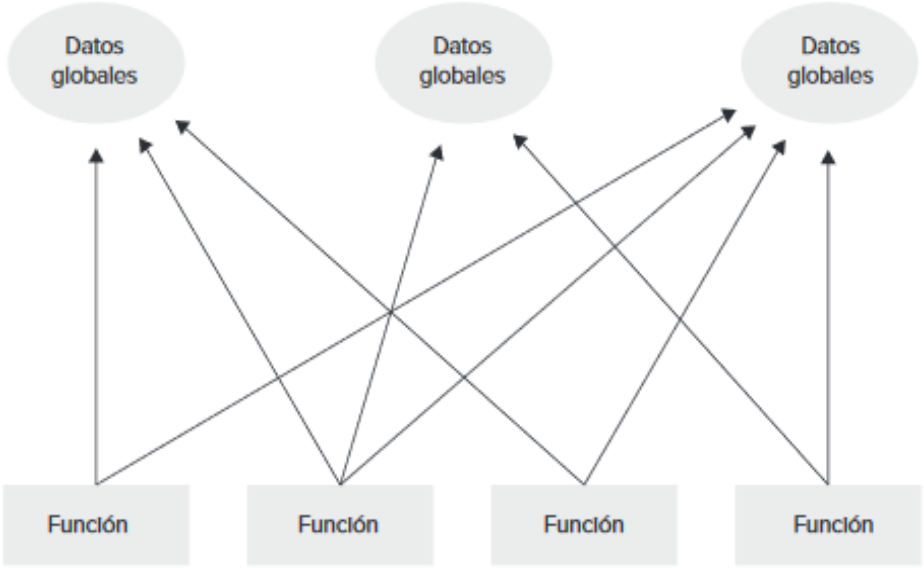


Figura 15.2. Un programa *procedimental.*

Cristian Pilapanta 1095211-2202112344

15.1.2. Modelado de objetos del mundo real: el camino a los objetos

La otra limitación de la programación estructurada reside en el hecho de que la separación de los datos y las funciones que los manipulan proporcionan un modelo muy pobre de las cosas y objetos del mundo real. En el mundo real se trata con objetos tales como personas, casas o motocicletas, y tienen a su vez incorporados *atributos (datos)* y *comportamiento (funciones).* Los objetos —complejos o no complejos— del mundo real tienen atributos y comportamiento.

Los **atributos** o características son las propiedades de los objetos; por ejemplo, para las personas, la estatura, el color del cabello y de los ojos, y la edad, entre otros; para un automóvil (coche o carro), la marca, la potencia, el número de puertas, el precio, etcétera. Los atributos del mundo real son equivalentes a los datos de un programa y tienen un valor determinado, 200 metros cuadrados, 20000 dólares, cinco puertas, etcétera.

Atributos = Datos

```
Persona = color del cabello (moreno)
          color de los ojos (castaños)
          estatura (1.83 m)
```

El **comportamiento** es la acción que realizan los objetos del mundo real en respuesta a un determinado estímulo. Por ejemplo, si se acelera un coche (carro), aumenta su velocidad; si se frena se ralentiza o se detiene. El comportamiento es similar a una función; la llamada a una función para realizar una tarea determinada, por ejemplo, dibujar un rectángulo o visualizar la nómina de los empleados de una empresa

Comportamiento = Funciones

Por estas razones, ni los datos ni las funciones, por sí mismas, modelan los objetos del mundo real de un modo eficiente y es la programación orientada a objetos el mejor método para modelar aplicaciones reales. La idea fun-

damental de la programación orientada a objetos es combinar en una sola entidad tanto los datos como las funciones que actúan sobre los datos. Tal unidad se denomina **objeto**. Esta característica permite modelar los objetos del mundo real de un modo mucho más eficiente que utilizando funciones y datos.

La *programación estructurada* mejora la claridad, fiabilidad y facilidad de mantenimiento de los programas; sin embargo, para programas grandes o a gran escala, presentan retos de difícil solución, que pueden ser resueltos más fácilmente mediante *programación orientada a objetos*.

15.2. Programación orientada a objetos

Al contrario que el enfoque procedimental que se basa en la interrogante *¿qué hace este programa?*, el enfoque orientado a objetos responde a otro interrogante *¿qué objetos del mundo real puede modelar?*

La POO (Programación orientada a objetos) se basa en el hecho de que se debe dividir el programa, no en tareas, sino en modelos de objetos físicos o simulados. Aunque esta idea parece abstracta a primera vista, se vuelve más clara cuando se consideran objetos físicos en términos de sus *clases*, *componentes*, *propiedades* y *comportamiento*, y sus objetos instanciados o creados de las clases.

Si se escribe un programa de computadora en un lenguaje orientado a objetos, se está creando, en su computadora, un modelo de alguna parte del mundo. Las partes que el modelo construye son los objetos que aparecen en el dominio del problema. Estos objetos deben ser representados en el modelo que se está creando en la computadora. Los objetos se pueden agrupar en categorías, y una clase describe —de un modo abstracto— todos los objetos de un tipo o categoría determinada.

Cristian Pilapaña 1095211-2202112344

Los objetos en C++, o en Java, modelan objetos del problema en el dominio de la aplicación.

Los objetos se crean a partir de las clases. La clase describe el tipo del objeto; los objetos representan instanciaciones individuales de la clase.

La idea fundamental de la orientación a objetos y de los lenguajes que implementan este paradigma de programación es combinar (encapsular) en una única unidad tanto los datos como las funciones que operan (manipulan) sobre los datos. Esta característica permite modelar los objetos del mundo real de un modo mucho más eficiente que con funciones y datos. Esta unidad de programación se denomina **objeto**.

Las funciones de un objeto, se llaman *funciones miembro* (en C++) o *métodos* (Java y otros lenguajes de programación), constituyen el único método para acceder a sus datos. Si se desea leer datos de un objeto se llama a una función miembro del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente ya que están ocultos y se dice que junto con las funciones están encapsulados en una entidad única. La *encapsulación* o *encapsulamiento* de los datos y la *ocultación* de los datos son conceptos clave en programación orientada a objetos.

La modificación de los datos de un objeto se realiza a través de una de las funciones miembro de ese objeto que interactúa con él, y ninguna otra función puede acceder a ellos. Esta propiedad facilita la escritura, depuración y mantenimiento de un programa.

En un sistema orientado a objetos, un programa se organiza en un conjunto finito de objetos que contienen datos y operaciones (*funciones miembro* o *método)* que se comunican entre sí mediante *mensajes* (llamadas a funciones miembro). La estructura de un programa orientado a objetos se muestra en la figura 15.3.

Las etapas necesarias para modelar un sistema —resolver en consecuencia un problema— empleando orientación a objetos son:

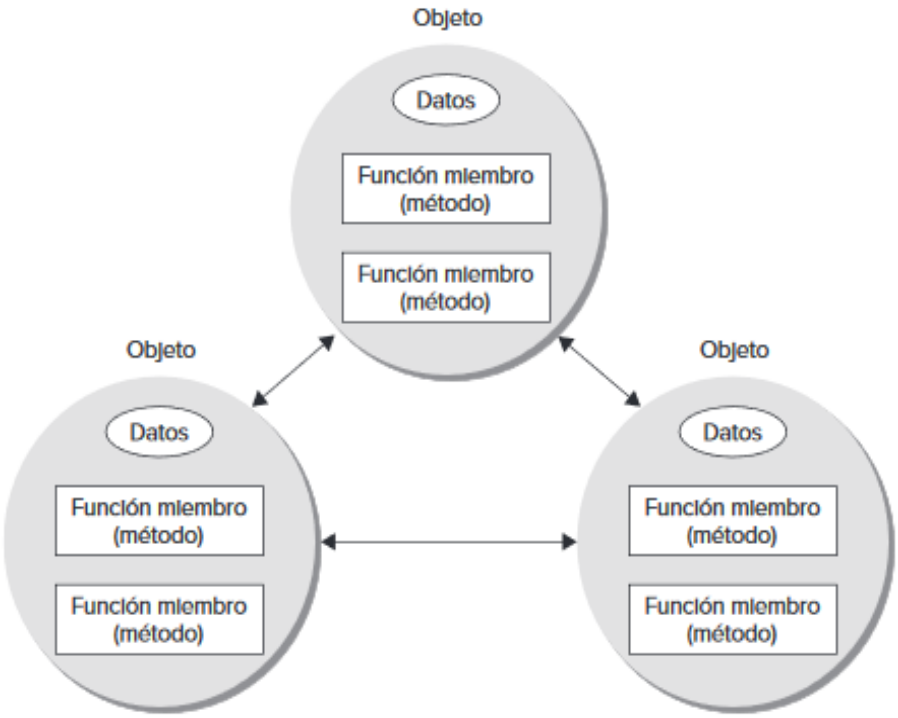


Figura 15.3. Organización típica de un programa orientada a objetos.

1. Identificación de los objetos del problema.
2. Agrupamiento en *clases* (tipos de objetos) de los objetos con características y comportamiento comunes.
3. Identificación de los datos y operaciones de cada una de las clases.
4. Identificación de las *relaciones* existentes entre las diferentes clases del modelo.

Los objetos encapsulan datos y funciones miembro o métodos que manipulan los datos. Las *funciones miembro* también se conocen como *métodos* —dependiendo de los lenguajes de programación—, por esta razón suelen ser términos sinónimos. Los elementos dato de un objeto se conocen también como *atributos* o *variables de instancia* (ya que instancia es un objeto específico). La llamada a una función miembro de un objeto se conoce como *envío de un mensaje* al objeto.

Objeto
- Funciones miembro o métodos.
- Atributos o variables de instancia (*datos*).

Envío de un mensaje a un objeto
- Llamada a la función miembro (*mensaje*).

15.2.1. Objetos

Una forma de reducir la complejidad es, como se verá más en profundidad en los siguientes apartados y capítulos, la *abstracción*. Las características y procesos de cualquier sistema se resumen en los aspectos esenciales y más relevantes; de este modo, las características complejas de los sistemas se vuelven más manejables.

En computación, la abstracción es el proceso crucial de representar la información en términos de su interfaz con el usuario. Es decir, se abstraen las características operacionales esenciales de un problema y expresa su solución

Cristian Pilapanta 1095211-2202112344

en dichos términos. *La abstracción se manifiesta en C++ con el diseño de una clase que implementa la interfaz y que no es más que un tipo de dato específico.*

Un ejemplo de abstracción expresada de diferentes formas según la aplicación a desarrollar, puede ser el término o clase `auto` (o bien coche, `carro`):

- Un `auto` es la composición o combinación de diferentes partes (`motor`, `cuatro ruedas`, `3 o 5 puertas`, `asientos`, entre otros).
- Un `auto` también es un término común que define a tipos diferentes de automóviles; se pueden clasificar por el fabricante (`BMW`, `Seat`, `Chevrolet`, `Toyota...`), por su categoría (o uso) (`deportivo`, `todo-terreno`, `sedán`, `pick-up`, `limousine`, `coupé`, etcétera).

Si los miembros de autos (automóviles o coches) o las diferencias entre coches individuales no son relevantes, se utiliza el término *coche* (o *carro*, en Latinoamérica) y entonces se utilizan expresiones y operaciones tales como: *se fabrican coches, se usan coches para ir de la Ciudad de México a Guadalajara, se descompone el coche en sus partes...*

El **objeto** es el centro de la programación orientada a objetos; un objeto es algo que se visualiza, se utiliza y que juega un papel o un rol. Cuando se programa de modo orientado a objetos se trata de descubrir e implementar los objetos que juegan un rol en el dominio del problema del programa. La estructura interna y el comportamiento de un objeto, en consecuencia, no es prioritario durante el modelado del problema. Es importante considerar que un objeto, tal como un `auto` juega un rol o papel importante.

Dependiendo del problema, diferentes aspectos de un objeto son significativos. Así, los *atributos* indican propiedades de los objetos: `propietario`, `marca`, `año de matriculación`, `potencia`, etcétera. El objeto también tiene funciones que actuarán sobre los atributos o datos; `matricular`, `comprar`, `vender`, `acelerar`, `frenar`, entre otras.

Un objeto no tiene que ser necesariamente algo concreto o tangible, puede ser totalmente abstracto y puede también describir un proceso. Un equipo de baloncesto puede ser considerado como un objeto, los atributos pueden ser los jugadores, el color de sus camisetas, partidos jugados, tiempo de juego, etcétera. Las clases con atributos y funciones miembro permiten gestionar los objetos dentro de los programas.

Cristian Pilaparta 1095211-2202112345

15.2.2. Tipos abstractos de datos: CLASES

Un progreso importante en la historia de los lenguajes de programación se produjo cuando se comenzó a combinar juntos diferentes elementos de datos y, por consiguiente, encapsular o empaquetar diferentes propiedades en un tipo de dato. Estos tipos fueron las *estructuras* o *registros* que permiten a una variable contener datos que pertenecen a las circunstancias representadas por ellas.

Las estructuras representan un modo de abstracción con los programas, concretamente la combinación (o *composición*) de partes diferentes o elementos (*miembros*). Así, por ejemplo, una estructura coche (carro) constará de miembros tales como marca, motor, número de matrícula, año de fabricación, entre otros.

Sin embargo, aunque en las estructuras y registros se pueden almacenar las propiedades individuales de los objetos en los miembros, en la práctica como están organizados, no pueden representar qué se puede hacer con ellos (moverse, acelerar, frenar, etc. en el caso de un coche/carro). Se necesita que las operaciones que forman la interfaz de un objeto se incorporen también al objeto.

El *tipo abstracto de datos* (*TAD*) describe, no sólo los atributos de un objeto sino también su comportamiento (*operaciones* o *funciones*) y, en consecuencia, se puede incluir una descripción de los estados que puede tener el objeto.

Así, un objeto "`equipo de baloncesto`" no sólo puede describir a los jugadores, la puntuación, el tiempo transcurrido, el periodo de juego, etc., sino que también se puede representar *operaciones* tales como "`sustituir un jugador`", "`solicitar tiempo muerto`", ..., o *restricciones* como, `el momento en que comienza 0:00 y en que termina cada cuarto de juego 15:00, o incluso las paradas del tiempo, por lanzamientos de personales.`

El término tipo abstracto de dato se consigue en programación orientada a objetos con el término *clase*. Una **clase** es la implementación de un tipo abstracto de dato y describe no sólo los *atributos* (datos) de un objeto, sino también sus *operaciones* (comportamiento). Así, la clase `carro` define que un coche/carro consta de motor, ruedas, placa de matrícula, entre otros, y se puede conducir (manejar), acelerar, frenar, etcétera.

Instancias

Una clase describe un objeto, en la práctica múltiples objetos. En conceptos de programación una clase es, realmente, un tipo de dato, y se pueden crear, en consecuencia, variables de ese tipo. En programación orientada a objetos a estas variables se les denomina *instancias* (*instances*), y también por sus sinónimos *ejemplares*, *casos*, etcétera.

Las instancias son la implementación de los objetos descritos en una clase. Estas instancias constan de los datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas en la propia clase.

En un lenguaje de programación OO, objeto e instancia son términos sinónimos. Así, cuando se declara una variable de tipo `Auto`, se crea un objeto `Auto` (una instancia de la clase `Auto`).

Métodos

En programación orientada a objetos, las operaciones definidas para los objetos, se denominan —como ya se ha comentado— métodos. Cuando se llama a una operación de un objeto se interpreta como el envío de un *mensaje* a dicho objeto.

Un programa orientado a objetos se forma enviando mensajes a los objetos, que a su vez producen (envían) más mensajes a otros objetos. Así, cuando se llama a la operación `"conducir"` (*manejar*) para un objeto auto en realidad lo que se hace es enviar el mensaje `"conducir"` al objeto `auto`, que procesa (ejecuta), a continuación, el método correspondiente.

En la práctica, un programa orientado a objetos es una secuencia de operaciones de los objetos que actúan sus propios datos.

Un **objeto** es una **instancia** o ejemplar de una **clase** (categoría o tipo de datos). Por ejemplo, un alumno y un profesor, son instancias de la clase `Persona`. Un objeto tiene una *estructura*, como ya se ha comentado; es decir, tiene *atributos (propiedades)* y *comportamiento*. El comportamiento de un objeto consta de operaciones que se ejecutan; y los atributos y las *operaciones* se llaman *características* del objeto.

EJEMPLOS

Un objeto de la clase `Persona` puede tener estos atributos: `altura`, `peso` y `edad`. Cada uno de estos atributos es único ya que son los valores específicos que tiene cada persona. Se pueden ejecutar estas operaciones: `dormir`, `leer`, `escribir`, `hablar`, `trabajar`, `correr`, entre otros. En C++, o en Java, estas operaciones dan lugar a las funciones miembro, o métodos: `dormir()`, `leer()`, `escribir()`, `hablar()`, `trabajar()`, `correr()`, etc. Si tratamos de modelar un sistema académico con profesores y alumnos, éstas y otras operaciones y atributos pertenecerán a dichos objetos y clases.

En el mundo de la orientación a objetos, una clase sirve para otros propósitos que los indicados de clasificación o categoría; una clase es una plantilla (tipo de dato) para hacer o construir objetos; por ejemplo, una clase `Lavadora` puede tener los atributos `nombreMarca`, `númeroSerie`, `capacidad` y `potencia`; y las operaciones `encender()` `prender` –, `apagar()`, `lavar()`, `aclarar()`...

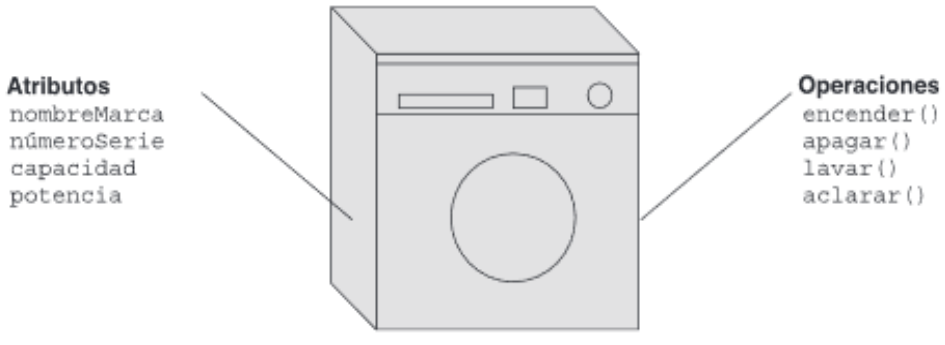


Figura 15.4. Notación gráfica de la clase `Lavadora` en UML.

Cristian Pilapanta 1095211-2202112345

Un anticipo de reglas de notación en UML 2.5.1

(*En el capítulo 16 se describe de manera más detallada la notación UML.*)

- El nombre de la clase comienza con una letra mayúscula (`Lavadora`).
- El nombre de la clase puede tener varias palabras y todas comenzar con mayúsculas, por ejemplo `PáginaWeb`.
- El nombre de una característica (atributo u operación) comienza con una letra minúscula (`estatura`).
- El nombre de una característica puede constar a su vez de dos palabras, en este caso la primera comienza con minúscula y la segunda con mayúscula (`nombreMarca`).
- Un par de paréntesis sigue al nombre de una operación; por ejemplo, `limpiar ()`.

Esta notación facilita el añadido o supresión de características, por lo cual representan gráficamente con mayor precisión un modelo del mundo real; por ejemplo, a la clase `Lavadora` se le podría añadir `velocidadMotor`, `volumen`, `aceptarDetergente()`, `ponerPrograma()`, y demás.

15.3. Modelado e identificación de objetos

En programación, un objeto es una entidad individual de un sistema que guarda una relación directa con los objetos del mundo real; la correspondencia entre ambos es el resultado práctico de combinar atributos y operaciones, o datos y funciones. Un objeto tiene un *estado*, un *comportamiento* y una *identidad*.

Cristian Pilapanta 1095211-2202112345

Estado

Conjunto de valores de todos los atributos de un objeto en un instante determinado. El estado de un objeto está determinado por los valores que toman sus datos o atributos. Estos valores han de cumplir siempre las restricciones (*invariantes de clase*, para objetos pertenecientes a la misma clase) que se hayan impuesto. El estado de un objeto tiene un carácter dinámico que evoluciona con el tiempo, con independencia de que ciertos elementos del objeto puedan permanecer constantes.

Comportamiento

Conjunto de operaciones que se pueden realizar sobre un objeto; éstas pueden ser de *observación* del estado interno del objeto, o bien de *modificación* de dicho estado. El estado de un objeto puede evolucionar en función de la aplicación de sus operaciones; éstas se realizan tras la recepción de un *mensaje* o estímulo externo enviado por otro objeto. Las interacciones entre los objetos se simbolizan mediante *diagramas de objetos*. En UML se representarán por enlaces en ambas direcciones.

Identidad

Permite diferenciar los objetos de modo no ambiguo independientemente de su estado. Es posible distinguir dos objetos en los cuales todos sus atributos sean iguales. Cada objeto posee su propia identidad de manera implícita; cada uno ocupa su propia posición en la memoria de la computadora.

Figura 15.5. Comunicación entre objetos por mensajes.

En un sistema orientado a objetos los programas se organizan en conjuntos finitos que contienen atributos (datos) y operaciones (funciones) y que se comunican entre sí mediante mensajes. Los pasos típicos en el modelado de un sistema orientado a objetos son:

1. Identificar los objetos que forman parte del modelo.
2. Agrupar en *clases* todos aquellos objetos que tengan características y comportamientos comunes.
3. Identificar los *atributos* y las *operaciones* de cada clase.
4. Identificar las *relaciones* existentes entre las clases.

Cuando se diseña un problema en un lenguaje orientado a objetos, se debe pensar en dividir el problema en objetos, o dicho de otro modo, es preciso identificar y seleccionar los objetos del dominio del problema de modo que exista una correspondencia entre los objetos desde el punto de vista de programación y los objetos del mundo real.

¿Qué tipos de cosas se convierten en objetos en los programas orientados a objetos? La realidad es que la gama de casos es infinita y sólo dependen de las características del problema a resolver y la imaginación del programador. Algunos casos típicos son:

Personas	***Partes de una computadora***	***Estructuras de datos***
Empleados	Pantalla	Pilas
Estudiantes	Unidades de CD, DVD *pendrive*	Colas
Clientes	Impresora	Listas enlazadas
Profesores	Teclado	Árboles
Edición de revistas	***Objetos físicos***	***Archivos de datos***
Artículos	Autos	Diccionario
Nombres de autor	Mesas	Inventario
Volumen	Carros	Listas
Número	Ventanas	Ficheros
Fecha de publicación	Farolas	Almacén de datos

La correspondencia entre objetos de programación y objetos del mundo real se muestra en una combinación entre datos y funciones.

15.4. Propiedades fundamentales de orientación a objetos

Los conceptos fundamentales de orientación a objetos que a su vez se constituyen en reglas de diseño en un lenguaje de programación orientado a objetos son: *abstracción, herencia (generalización), encapsulamiento, ocultación de datos, polimorfismo y reutilización*. Otras propiedades importantes son: *envío de mensajes* y diferentes tipos de relaciones tales como, *asociaciones y agregaciones* (véase el capítulo 17).

15.4.1. Abstracción

La abstracción es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. En computación, la abstracción es la etapa crucial de representación de la información en términos de la interfaz con el usuario. La abstracción se representa con un tipo definido por el usuario, con el diseño de una clase que implementa la interfaz correspondiente. Una *clase* es un elemento en C++ o en Java, que traduce una abstracción a un tipo definido por el usuario. Combina representación de datos y métodos para manipular esos datos en un paquete.

La abstracción posee diversos grados denominados *niveles de abstracción* que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el análisis de un sistema hay que concentrarse en *¿qué hace?* y no en *¿cómo lo hace?*

El principio de la *abstracción* es más fácil de entender con una analogía del mundo real. Por ejemplo, una televisión es un aparato electrodoméstico que se encuentra en todos los hogares. Seguramente, estará familiarizado con sus características y su manejo manual o con el mando a distancia: encender (prender), apagar, cambiar de canal, ajustar el volumen, cambiar el brillo..., y añadir componentes externos como altavoces, grabadoras de CD, repro-

Cristian Pilaparita 1095211-2202112345

ductoras de DVD, conexión de un módem para Internet, etcétera. Sin embargo, ¿sabe usted cómo funciona internamente?, ¿conoce cómo recibe la señal por la antena, por cable, por satélite, traduce la señal y la visualiza en pantalla? Normalmente sucede que *no* sabemos cómo funciona el aparato de televisión, pero *sí* sabemos cómo utilizarlo. Esta característica se debe a que la televisión separa claramente su *implementación interna* de su *interfaz externa* (el cuadro de mandos de su aparato o su mando a distancia). Actuamos con la televisión a través de su interfaz: los botones de alimentación, de cambio de canales, control de volumen, entre otros. No conocemos el tipo de tecnología que utiliza, el método de generar la imagen en la pantalla o cómo funciona internamente, es decir su *implementación*, ya que ello no afecta a su interfaz.

15.4.2. La abstracción en el *software*

El principio de abstracción es similar en el software. Se puede utilizar código sin conocimiento de la implementación fundamental. En C++, por ejemplo, se puede hacer una llamada a la función `sqrt()`, declarada en el archivo de cabecera `<math.h>`, que puede utilizar sin necesidad de conocer la implementación del algoritmo real que calcular la raíz cuadrada. De hecho, la implementación fundamental del cálculo de la raíz cuadrada puede cambiar en las diferentes versiones de la biblioteca, mientras que la interfaz permanece tal cual.

También se puede aplicar el principio de abstracción a las clases. Así, se puede utilizar el objeto `cout` de la clase `ostream` para *fluir* (enviar) datos a la salida estándar, en C++, como

```
cout << " En un lugar de Cazorla \n";
```

o en pseudocódigo

```
escribir "En un lugar de Cazorla" <fin de línea>
```

En la línea anterior, se utiliza la interfaz documentada del operador de inserción `cout` con una cadena, pero no se necesita comprender cómo `cout` administra visualizar el texto en la interfaz del usuario. Sólo necesita conocer la interfaz pública. Así, la implementación fundamental de cout es libre de cambiar, mientras que el comportamiento expuesto y la interfaz permanecen iguales.

La abstracción es el principio fundamental que se encuentra tras la *reutilización*. Sólo se puede reutilizar un componente si en él se ha abstraído la esencia de un conjunto de elementos del confuso mundo real que aparece una y otra vez, con ligeras variantes en sistemas diferentes.

15.4.3. Encapsulamiento y ocultación de datos

La **encapsulación o encapsulamiento**, significa reunir en una estructura a todos los elementos que a un cierto nivel de abstracción se pueden considerar pertenecientes a una misma entidad, y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que permite aumentar la cohesión de los componentes del sistema.

En este caso, los objetos que poseen las mismas características y comportamiento se agrupan en clases que son unidades de programación que encapsulan datos y operaciones.

La encapsulación oculta lo *que hace* un objeto de *lo que hacen* otros objetos y del mundo exterior, por lo que se denomina también *ocultación de datos*.

Un objeto tiene que presentar "una cara" al mundo exterior de modo que se puedan iniciar esas operaciones. El aparato de TV tiene un conjunto de botones, bien en la propia TV o incorporados en un mando a distancia. Una máquina lavadora tiene un conjunto de mandos e indicadores que establecen la temperatura y el nivel del agua. Los botones de la TV y las marchas de la máquina lavadora constituyen la comunicación con el mundo exterior, las interfaces.

En esencia, la *interfaz* de una clase representa un "contrato" de prestación de servicios entre ella y los demás componentes del sistema. De este modo, los clientes de un componente sólo necesitan conocer los servicios que éste ofrece y no cómo están implementados internamente.

Por consiguiente, se puede modificar la implementación de una clase sin afectar a las restantes relacionadas con ella. En general, esto implica mantener el contrato y se puede modificar la implementación de una clase sin afectar a las restantes clases relacionadas con ella; sólo es preciso mantener el contrato. Existe una separación de la

Cristian Pilapanta 1095211-2202112345

interfaz y de la implementación. La interfaz pública establece qué se puede hacer con el objeto; de hecho, la clase actúa como una "caja negra". La interfaz pública es estable, pero la implementación se puede modificar.

15.4.4. Herencia

El concepto de clase conduce al de herencia. En la vida diaria se utiliza el concepto de *clases* que se dividen a su vez en *subclases*. La clase `animal` se divide en `mamíferos`, `anfibios`, `insectos`, `pájaros`, etcétera. La clase `vehículo` se divide en `autos`, `coches`, `carros`, `camiones`, `buses`, `motos`, entre otros. La clase `electrodoméstico` se divide en `lavadora`, `frigorífico`, `tostadora`, `microondas`, etcétera.

La idea principal de estos tipos de divisiones reside en el hecho de que cada subclase comparte características con la clase de la cual se deriva. Los autos, camiones, buses, motos..., tienen motor, ruedas y frenos. Además de estas características compartidas, cada subclase tiene sus propias características. Los autos, por ejemplo, pueden tener maletero, cinco asientos; los camiones cabina y caja para transportar carga, y demás.

La clase principal de la que derivan las restantes se denomina *clase base* –en C++–, *clase padre* o *superclase* y las *subclases*, también se denominan *clases derivadas* –en C++–.

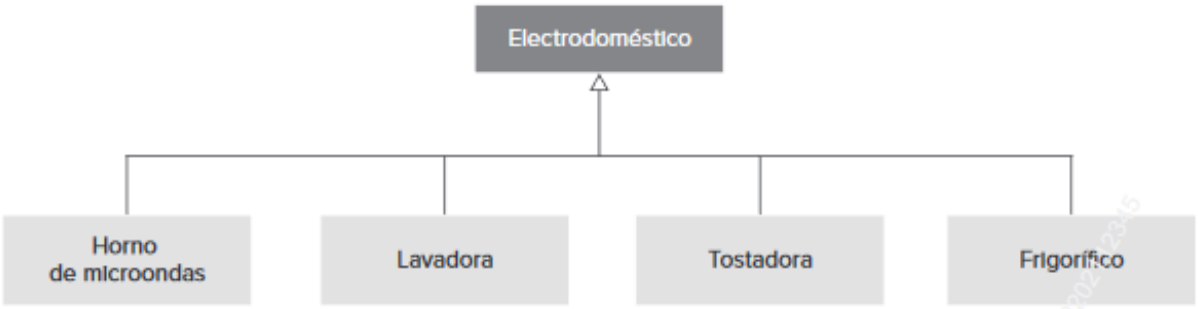


Figura 15.6. Herencia simple.

Las clases modelan el hecho de que el mundo real contiene objetos con propiedades (*atributos*) y comportamiento. La herencia modela el hecho de que estos objetos tienden a organizarse en jerarquías. Esta jerarquía desde el punto de vista del modelado se denomina relación de *generalización* o ***es-un*** **(*is-a*)**. En programación orientada a objetos, la relación de generalización se denomina *herencia.* Cada clase derivada hereda las características de la clase base y además cada clase derivada añade sus propias características (atributos y operaciones). Las clases bases pueden a su vez ser también subclases o clases derivadas de otras superclases o clases base.

Así, el programador puede definir una clase `Animal` que encapsula todas las propiedades o atributos (`altura`, `peso`, `número de patas`, etc.) y el comportamiento u operaciones (`comer`, `dormir`, `andar`) que pertenecen a cada animal. Los animales específicos como un `Mono`, `Jirafa`, `Canguro` o `Pingüino` tienen a su vez cada uno de ellos características propias.

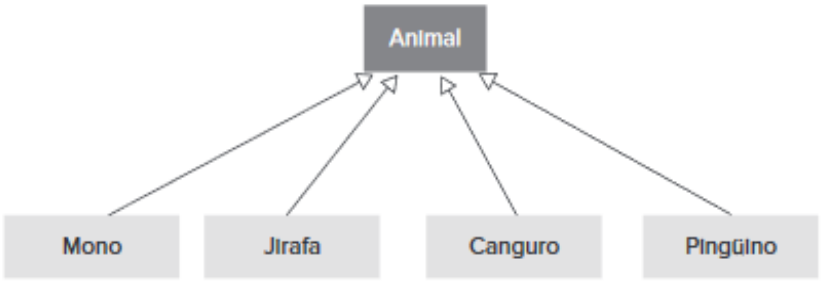


Figura 15.7. Herencia de la clase `Animal`.

Las técnicas de herencia se representan con la citada relación ***es-un***. Así, se dice que un `Mono` `es-un` `Animal` que tiene características propias: *puede subir a los* árboles, *saltar de un* árbol *a otro,* etc.; además tiene en común con la `Jirafa`, el `Canguro` y el `Pingüino`, las características propias de cualquier animal (*comer, beber, correr, dormir,...*).

Cristian Pilaparta 1095211-2202112345

15.4.5. Reutilización o *reusabilidad*

Otra propiedad fundamental de la programación orientada a objetos es la **reutilización** o ***reusabilidad***.[1] Este concepto significa que una vez que se ha creado, escrito y depurado una clase, se puede poner a disposición de otros programadores. El concepto es similar al uso de las bibliotecas de funciones en un lenguaje de programación procedimental como C.

El concepto de herencia en C++ o Java, proporciona una ampliación o extensión importante a la idea de *reusabilidad*. Una clase existente se puede ampliar añadiéndole nuevas características (atributos y operaciones). Esta acción se realiza derivando una nueva clase de la existente. La nueva clase hereda las características de la clase base; pero podrá añadirle nuevas características.

Reutilización de código

La facilidad de reutilizar o *reusar* el software existente es uno de los grandes beneficios de la POO. De este modo en una empresa de software se pueden reutilizar clases diseñadas en un proyecto en un nuevo proyecto con la consiguiente mejora de la productividad, al sacarle partido a la inversión realizada en el diseño de la clase primitiva.

En esencia, ¿cuáles son las ventajas de la herencia? Primero, reducción de código; las otras propiedades comunes de varias clases sólo necesitan ser implementadas una vez y si es necesario sólo deben modificarse una vez. Segundo, está soportado el concepto de abstracción de la funcionalidad común.

La facilidad con la que el software existente se puede reutilizar es una propiedad muy importante de la POO. La idea de usar código existente no es nueva en programación, ni en C++ ni Java. Cada vez que se imprime algo con `cout` se está reutilizando código. No se escribe el código de salida a pantalla para visualizar los datos sino que se utiliza el flujo `ostream` existente para realizar el trabajo. Muchas veces los programadores no tienen en cuenta la ventaja de utilizar el código disponible y, como suele decirse en la jerga de la programación, "inventan la rueda" cada día, o al menos en cada proyecto.

Por ejemplo, supongamos que usted desea ejecutar un planificador de un sistema operativo. El planificador *(scheduler)* es un componente del SO responsable de planificar procesos (cuáles se deben ejecutar y durante cuánto tiempo). Para realizar esta planificación suele recurrirse a implementaciones basadas en prioridades, para lo cual se requiere una estructura de datos denominada *cola de prioridades* que almacena los procesos a la espera de que se vayan ejecutando en función de su prioridad. Existen en C++ dos métodos: 1) escribir el código correspondiente a una *cola de prioridades*; 2) la biblioteca estándar de plantillas STL incorpora un contenedor denominado *priorty-queue* (cola de prioridad) que puede utilizarse para almacenar objetos de cualquier tipo. Lo más sensato es incorporar en su programa el contenedor `priority-queue` de la biblioteca STL en un diseño del planificador, en lugar de reescribir su propia cola de prioridad.

Reescritura de código reusable

Es importante en el diseño de un código la escritura de *código reusable*. Debe diseñar sus programas de modo que pueda reutilizar sus clases, sus algoritmos y sus estructuras de datos. Tanto usted como sus colegas en el proyecto en el que trabaje deben poder utilizar los componentes anteriores, tanto en el proyecto actual como en futuros proyectos. En general, se debe evitar diseñar código en exceso o específico para casos puntuales; *siempre que sea posible reutilice código existente.*

Una técnica del lenguaje C++ para escribir código de propósito general es utilizar *plantillas* (*templates*). En lugar de escribir, por ejemplo, una estructura de datos `Pila` para tratar números reales, es preferible diseñar una plantilla o tipo genérico `Pila` que le sirva para cualquier tipo de dato a procesar.

15.4.6. Polimorfismo

El **polimorfismo** es la propiedad que permite a una operación (función) tener el mismo nombre en clases diferentes y que actúe de modo diferente en cada una de ellas. Esta propiedad es intrínseca a la vida ordinaria ya que una

[1] El término ***reusabilidad*** no figura todavía en la 23a. edición del Diccionario de la Real Academia Española (DRAE); sin embargo, si está incorporado el término **reutilización**: "acción y efecto de reutilizar". El término reusable tampoco está en la 23a. edición del DRAE.

Cristian Pilapanta 1095211-2202112345

misma operación puede realizar diferentes acciones dependiendo del *objeto* sobre el que se aplique. Así, por ejemplo, se puede abrir una puerta, abrir una ventana, abrir un libro, abrir un periódico, abrir una cuenta corriente en un banco, abrir una conversación, abrir un congreso, etcétera. En cada caso se realiza una operación diferente. En orientación a objetos, cada clase "conoce" cómo realizar esa operación.

En la práctica, el polimorfismo significa la capacidad de una operación de ser interpretada sólo por el propio objeto que lo invoca. Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente qué operación ha sido invocada.

En la vida diaria hay numerosos ejemplos de polimorfismo. En un taller de reparaciones de automóviles existen numerosos automóviles de marcas diferentes, de modelos diferentes, de potencias diferentes, de carburantes diferentes, etcétera. Constituyen una clase o colección heterogénea de carros (coches). Supongamos que se ha de realizar una operación típica "cambiar los frenos del carro". La operación a realizar es la misma, incluye los mismos principios de trabajo, sin embargo, dependiendo del coche, en particular, la operación será muy diferente, incluirá diferentes acciones en cada caso.

La propiedad de polimorfismo, en esencia, es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. El polimorfismo es importante tanto en el modelado de sistemas como en el desarrollo de software. En el modelado porque el uso de palabras iguales tiene comportamientos distintos, según el problema a resolver. En software, porque el polimorfismo toma ventaja de la propiedad de la herencia.

En el caso de operaciones en C++ o Java, el polimorfismo permite que un objeto determine en tiempo de ejecución la operación a realizar. Supongamos, por ejemplo, que se trata de realizar un diseño gráfico para representar figuras geométricas, tales como triángulo, rectángulo y circunferencia, y que se desea calcular la superficie o el perímetro (longitud) de cada una. Cada figura tiene un método u operación distinta para calcular su perímetro y su superficie. El polimorfismo permite definir una única función `calcularSuperficie`, cuya implementación es diferente en la clase `Triángulo`, `Rectángulo` o `Circunferencia`, y cuando se selecciona un objeto específico en tiempo de ejecución, la función que se ejecuta es la correspondiente al objeto específico de la clase seleccionada.

En C++ y Java existe otra propiedad importante derivada del polimorfismo y es la **sobrecarga de operadores y funciones**, que es un tipo especial de polimorfismo. La sobrecarga básica de operadores existe siempre. El operador + sirve para sumar números enteros o reales, pero si desea sumar números complejos deberá sobrecargar el operador + para que permita realizar esta suma.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que están actuando se denomina polimorfismo (una cosa con diferentes formas). Sin embargo, cuando a un operador existente, tal como + o =, se le permite la posibilidad de operar con diferentes tipos de datos, se dice que dicho operador está sobrecargado. *La sobrecarga es un tipo especial de polimorfismo* y una característica sobresaliente de los lenguajes de programación orientada a objetos.

Cristian Pizarro | 159211-20211234

Lenguajes de programación orientada a objetos

Los lenguajes de programación orientada a objetos utilizados en la actualidad son diversos, y aunque la mayoría siguen criterios de terminología universales, puede haber algunas diferencias: *puros* (Smalltalk, Eiffel, ...) *híbridos* (**C++**, **Java**, **C#**, Python, Objective-C, Object Pascal, VB.NET, ...)

15.5. Modelado de aplicaciones: UML

El **Lenguaje Unificado de Modelado** (**UML**, *Unified Model Language*), es el lenguaje estándar de modelado para desarrollo de sistemas y de software y su popularidad ha crecido en el modelado de otros dominios. Tiene una gran aplicación en la representación y modelado de la información que se utiliza en las fases de análisis y diseño. En diseño de sistemas se modela por una importante razón: *gestionar la complejidad*.

Un modelo es una abstracción de cosas reales. Cuando se modela un sistema, se realiza una abstracción ignorando los detalles que sean irrelevantes. El modelo es una simplificación del sistema real. Con un lenguaje formal de mo-

delado, el lenguaje es abstracto y preciso como un lenguaje de programación. Esta precisión permite que un lenguaje sea legible por la máquina, de modo que pueda ser interpretado, ejecutado y transformado entre sistemas. Para modelar un sistema de modo eficiente, se necesita algo muy importante: un lenguaje que pueda describir el modelo. ¿Qué es UML? UML es un **lenguaje**, esto significa que tiene tanto sintaxis como semántica y se compone de: pseudocódigo, código real, dibujos, programas, descripciones. Los elementos que constituyen un lenguaje de modelado se denominan **notación**.

El bloque básico de construcción de UML es un *diagrama.* Existen tipos diferentes, algunos con propósitos muy específicos (*diagramas de tiempo*) y algunos con usos más genéricos (*diagramas de clases*).

Un lenguaje de modelado puede ser cualquier cosa que contiene una *notación* (un medio de expresar el modelo) y una *descripción* de lo que significa esa notación (un *meta-modelo*). Aunque existen diferentes enfoques y visiones de modelado, UML tiene un gran número de ventajas que lo convierten en un lenguaje idóneo para un gran número de aplicaciones como:

- Diseño de software.
- Software de comunicaciones.
- Proceso de negocios.
- Captura de detalles acerca de un sistema, proceso u organización en análisis de requisitos.
- Documentación de un sistema, proceso o sistema existente.

UML se ha aplicado y se sigue aplicando en un sinfín de dominios, como:

- Banca.
- Salud.
- Defensa.
- Computación distribuida.
- Sistemas empotrados.
- Sistemas en tiempo real.
- Etcétera.

Cristian Pilapanta 1095211-2202112345

1095211220211234S

15.5.1. Lenguaje de modelado

Un modelo es una simplificación de la realidad que se consigue mediante una abstracción. El modelado pretende capturar las partes fundamentales o esenciales de un sistema y se representa mediante una notación gráfica.

Un modelo se expresa en un **lenguaje de modelado** y consta de notación —símbolos utilizados en los modelos— y un conjunto de reglas que instruyen cómo utilizarlas. Las reglas son sintácticas, semánticas y pragmáticas.

- **Sintaxis**, nos indica cómo se utilizan los símbolos para representar elementos y cómo se combinan en el lenguaje de modelado. La sintaxis es comparable con las palabras del lenguaje natural; es importante conocer cómo se escriben correctamente y cómo poner los elementos para formar una sentencia.
- **Semántica**, las reglas semánticas explican lo que significa cada símbolo y cómo se deben interpretar, bien por sí misma o en el contexto de otros símbolos.
- **Pragmática**, las reglas de pragmática definen las intenciones de los símbolos a través de los cuales se consigue el propósito de un modelo y se hace comprensible para los demás. Esta característica se corresponde en lenguaje natural con las reglas de construcción de frases que son claras y comprensibles. Para utilizar bien un lenguaje de modelado es necesario aprender estas reglas.

Un modelo UML tiene dos características muy importantes:

- **Estructura estática**: describe los tipos de objetos más importantes para modelar el sistema.
- **Comportamiento dinámico**: describe los ciclos de vida de los objetos y cómo interactúan entre sí para conseguir la funcionalidad del sistema requerida.

Estas dos características están estrechamente relacionadas entre sí.

15.5.2. UML: el Lenguaje Unificado de Modelado

UML es el lenguaje de modelado estándar para desarrollo de software y de sistemas. Algunas preguntas importantes, a responder, para los desarrolladores son: ¿Por qué UML es unificado? ¿Qué se puede modelar? ¿Por qué UML es un lenguaje?

El diseño de sistemas de media y gran complejidad suele ser difícil. Desde una simple aplicación de escritorio/oficina hasta un sistema para la Web o para gestión de relación con clientes de una gran empresa, entraña construir centenares —o millares— de componentes de software y hardware. En realidad, la principal razón para modelar un diseño de sistemas es gestionar (administrar) la complejidad.

Un modelo es una *abstracción* de cosas del mundo real (tangibles o intangibles). El modelo es una simplificación del sistema real, de modo que facilite el diseño y la viabilidad de un sistema y que pueda ser comprendido, evaluado, analizado y criticado. En la práctica un lenguaje formal de modelado puede ser tan preciso como un lenguaje de programación. Esta precisión puede permitir que un lenguaje sea legible por la máquina, y pueda ser interpretado, ejecutado y transformado entre sistemas.

Para modelar eficazmente un sistema se necesita algo muy importante: un lenguaje que pueda describir con rigor el modelo. Un lenguaje de modelado se puede construir con pseudocódigo, código real, dibujos, diagramas, descripciones de texto, etc. Los elementos que constituyen un lenguaje de modelado se llaman *notación*.

En general, un modelo UML se compone de uno o más *diagramas*. Un diagrama gráfico representa cosas y relaciones entre las cosas. Estas cosas pueden ser representaciones de objetos del mundo real, construcciones de software o una descripción del comportamiento de algún otro objeto. En la práctica cada diagrama representa una *vista* de la cosa u objeto a modelar.

UML es un lenguaje de modelado visual para desarrollo de sistemas. La característica de extensibilidad hace que UML se pueda emplear en aplicaciones de todo tipo, aunque su fuerza y la razón por la que fue creado es modelar sistemas de software orientado a objetos dentro de áreas tales como programación e ingeniería de software.

UML como lenguaje de modelado visual es independiente del lenguaje de implementación, de modo que los diseños realizados usando UML se pueden implementar en cualquier lenguaje que soporte las características de UML, esencialmente lenguajes orientados a objetos como C++, C# o Java.

Un lenguaje de modelado puede ser todo aquel que contenga una notación (un medio de expresar el modelo) y una descripción de lo que significa esa notación (un metamodelo). ¿Qué ventajas aporta UML a otros métodos, lenguajes o sistemas de modelado? Las principales ventajas de UML son [Miles, Hamilton 06]:

- Es un *lenguaje formal,* cada elemento del lenguaje está rigurosamente definido.
- Es *conciso.*
- Es *comprensible, completo,* describe todos los aspectos importantes de un sistema.
- Es escalable, sirve para gestionar grandes proyectos pero también pequeños proyectos.
- *Está construido sobre la filosofía de "lecciones aprendidas",* ya que recogió lo mejor de 3 métodos ya muy probados y desde su primer borrador, en 1994, ha incorporado las mejores prácticas de la comunidad de orientación a objetos.
- Es un *estándar,* ya que está avalado por la OMG, organización con difusión y reconocimiento mundial.

15.6. Modelado y modelos

El enfoque principal de UML es el modelado. Sin embargo, ¿cuál es el significado de modelado?, ¿qué significa exactamente? Son preguntas abiertas que UML contesta adecuadamente. Modelado es un medio de capturar ideas, relaciones, decisiones y requerimientos en una notación bien definida que se puede aplicar a muchos dominios diferentes. Modelado no sólo significa cosas diferentes para personas distintas, sino también utiliza diferentes piezas dependiendo de lo que se está tratando de transmitir.

En general, un modelo en UML se construye de uno o más *diagramas.* Un diagrama gráficamente representa cosas, y las relaciones entre estas cosas. Estas cosas pueden ser representaciones de objetos del mundo real, construcciones de software puras, o una descripción del comportamiento de algún otro objeto. Es frecuente que una cosa individual se muestre en diagramas múltiples; cada diagrama representa un interés particular, o vista, de la cosa que se está modelando. UML consta de un número de elementos gráficos que se combinan para formar diagramas. Ya

Cristian Pilapanta 1095211-2202112345

que UML es un lenguaje, dispone de reglas para combinar estos elementos; su conocimiento será importante y también el conocimiento de los diagramas que le ayuden a diseñar el modelo.

El propósito de los diagramas es presentar múltiples vistas de un sistema; este conjunto de vistas múltiples se llama modelo. En la práctica un **modelo** es una descripción abstracta de un sistema o de un proceso, una representación simplificada que permite comprender y simular. Un modelo UML de un sistema es similar a un modelo en escala de un edificio junto con la representación artística del mismo. Es importante observar que un modelo UML de un sistema describe aquello que se supone hace. No indica cómo implementar el sistema. En el siguiente apartado se describen los diagramas fundamentales de UML.

El concepto de modelo es muy útil en numerosos campos, y en particular en los campos de la ciencia e ingeniería.

En el sentido más general, cuando se crea un modelo se está utilizando algo que se conoce en gran medida y trata de ayudar a comprender que ese algo se entienda mucho mejor. En algunos campos un modelo en un conjunto de ecuaciones; en otros, un modelo es una simulación computacional. Son posibles muchos tipos de modelos.

En nuestro caso, un modelo es un conjunto de diagramas que podemos examinar, evaluar y modificar con la finalidad de comprender y desarrollar un sistema. En ocasiones suele utilizarse el concepto de metamodelo como la descripción de modo formal de los elementos del modelado y la sintaxis y la semántica de la notación que permite su manipulación; en esencia, como ya se comentó anteriormente, es una descripción de la representación de las notaciones empleadas.

Un modelo es la unidad básica del desarrollo, se enlaza a una fase precisa y se construye a partir de elementos de modelado con sus diferentes vistas asociadas. El concepto de modelo es muy útil en numerosos campos, en particular, en ciencias e ingeniería.

En el campo de computación, un modelo es un conjunto de diagramas UML que se pueden examinar, evaluar y modificar con el objetivo de entender y desarrollar un sistema.

Cristian Pilaparia 1095211-2202112345

El término modelado se emplea a menudo como sinónimo de análisis, es decir, de descomposición en elementos simples, más fáciles de comprender. En computación (informática), el modelado consiste ante todo en describir un problema, luego en describir la solución de éste; estas actividades se conocen, respectivamente, como el *análisis* y el *diseño*.

Categorías de modelado

El término modelado expresa la descomposición en elementos simples más fáciles de comprender y diseñar. El modelado de un sistema se hace típicamente desde tres puntos de vista distintos:

- *Modelado de objetos* (descompone los sistemas en objetos colaboradores).
  Aspectos estáticos y estructurales del sistema.
- *Modelado dinámico* (describe los aspectos temporales "timeline" del sistema).
  Aspectos temporales y de comportamiento del sistema.
- *Modelado funcional* (descompone las tareas en funciones de realización más simples).
  Aspectos de transformación funcional del sistema.

Estos tres tipos generales de modelado se definen en UML por diferentes modelos que realizan la representación de los sistemas:

- Modelo de clases que captura la estructura estática.
- Modelo de estados que expresa el comportamiento dinámico de los objetos.
- Modelo de casos de uso que describe las necesidades del usuario.
- Modelo de interacción que representa los escenarios y los flujos de mensajes.
- Modelo de realización que muestra las unidades de trabajo.
- Modelo de despliegue que precisa el reparto de procesos.

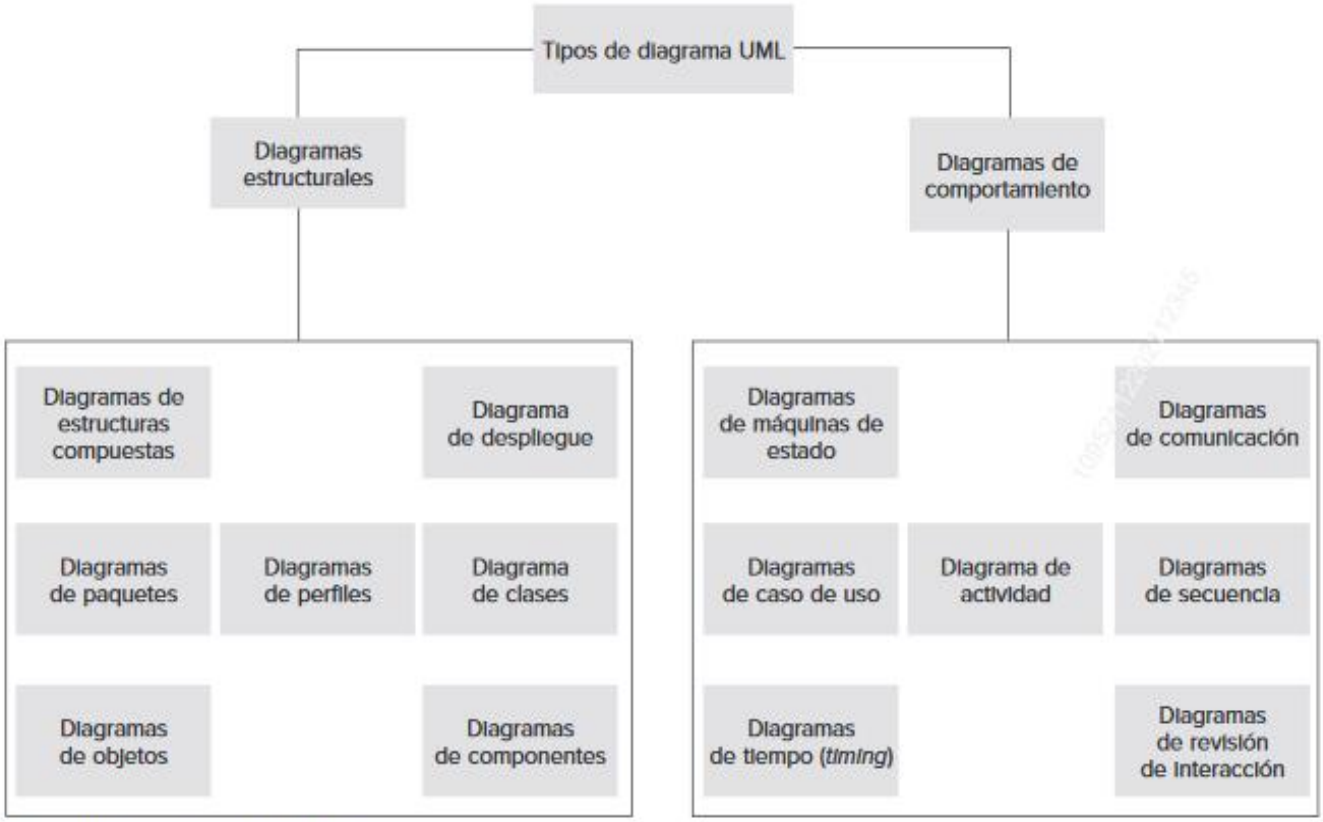
Los modelos se ven y manipulan por los usuarios mediante vistas gráficas, verdaderas proyecciones a través de los elementos de modelado contenidos por uno o varios modelos. Se pueden construir numerosas vistas a partir de los modelos básicos que pueden mostrar la totalidad o parte de los mismos. A cada vista corresponden uno o varios

diagramas. UML define 14 tipos de diagramas diferentes que veremos más adelante. UML es una representación gráfica normalizada de un modelo semántico común.

15.7. Diagramas de UML 2.5.1

UML es un lenguaje unificado de modelado y de propósito general y estandarizado (ISO/IEC 19501: 2005) se utiliza normalmente en ingeniería de software orientado a objetos, pero al ser un lenguaje muy rico, se puede utilizar para modelar estructuras de aplicaciones, comportamiento e incluso procesos de negocio.

Existen 14 tipos de diagramas en UML 2.5 que ayudan a modelar este comportamiento. La taxonomía de estos diagramas se divide en dos grandes categorías principales: diagramas de estructura (estructurales) y diagramas de comportamiento (figura 15.8).



Fuente: OMG y Creately.com (adaptada).

Figura 15.8. Taxonomía de diagramas (notaciones) UML 2.5.

Los *diagramas estructurados* o *estructurales* se utilizan para capturar la organización física de las cosas del sistema, por ejemplo, cómo se relacionan unos objetos con otros. Los diagramas estructurados son:

- Diagramas de clases.
- Diagramas de componentes.
- Diagramas de despliegue.
- Diagramas de objetos.
- Diagramas de paquetes.
- Diagramas de perfiles.
- Diagramas de estructuras compuestas.

Cristian Pilapaña 1095211-2202112345

El modelo estructural representa el marco de trabajo del sistema y este marco de trabajo es el lugar donde existen los demás componentes. De modo que los diagramas de clases, diagrama de componentes y diagramas de despliegue son las partes del modelado estructural; representan los elementos y mecanismos para ensamblarlos. Sin embargo, el modelo estructural nunca describe el comportamiento dinámico del sistema. El diagrama de clase es el diagrama estructural más ampliamente utilizado.

Los *diagramas de comportamiento* se centran en el comportamiento de los elementos de un sistema y describen la interacción en el sistema. Representan la interacción entre los diagramas estructurales. En resumen, el modelado de comportamiento muestra la naturaleza dinámica del sistema. Por ejemplo, se pueden utilizar diagramas de comportamiento para capturar requerimientos, operaciones y cambios de los estados internos de los elementos. Los diagramas de comportamiento son:

- Diagramas de actividad.
- Diagramas de comunicación.
- Diagramas de revisión de interacción.
- Diagramas de secuencia.
- Diagramas de máquinas de estado.
- Diagramas de tiempo (*timing*).
- Diagramas de caso de uso.

Diagramas de clase

Los diagramas de clase representan la estructura estática en términos de clases y relaciones; utilizan clases de interfaces para capturar detalles sobre las entidades que constituyen su sistema y las relaciones estáticas entre ellas. Los diagramas de clases son uno de los diagramas UML más utilizados en modelado y en la generación de código fuente en un lenguaje de programación.

Diagramas de componentes

Representan los componentes físicos de una aplicación; muestran la organización y dependencias implicadas en la implementación de un sistema. Pueden agrupar elementos más pequeños, como clases, en piezas más grandes y desplegables.

Diagramas de estructura compuesta

Los diagramas de estructura compuesta enlazan diagramas de clases y diagramas de componentes.

Diagramas de despliegue

Estos diagramas muestran cómo un sistema se ejecuta realmente o se asignan a varias piezas de hardware. Por lo general, los diagramas de despliegue muestran cómo los componentes se configuran en tiempo de ejecución y representan el despliegue de los componentes sobre los dispositivos materiales.

Diagramas de paquetes

Los diagramas de paquetes son realmente tipos especiales de diagramas de clases. Proporcionan un medio para visualizar dependencias entre partes de su sistema y se utilizan, a menudo, para examinar problemas o determinar el orden de compilación.

Diagramas de objetos

Estos diagramas representan los objetos y sus relaciones y corresponden a diagramas de colaboración simplificados, sin representación de los envíos de mensajes. Utilizan la misma sintaxis que los diagramas de clase y muestran cómo las instancias reales de clases se relacionan en una instancia específica en un momento dado. Los diagramas de clases muestran instantáneas de las relaciones de su sistema en tiempo de ejecución.

Cristian Pilaquinga 1095211-2202112345

Diagramas de actividad

Los diagramas de actividades representan el comportamiento de una operación en términos de acciones. Capturan el flujo de un comportamiento o actividad, al siguiente. Son similares, en concepto, a los clásicos diagramas de flujo pero de manera más expresiva.

Diagramas de comunicación

Estos diagramas son un tipo de diagrama de interacción que se centra en los elementos implicados en un comportamiento particular y cuáles mensajes se pasan en uno y otro sentido. Los diagramas de comunicación enfatizan más en los objetos implicados que en el orden y naturaleza de los mensajes intercambiados.

Diagramas de descripción de la interacción

Estos diagramas son versiones simplificadas de diagramas de actividad. En lugar de enfatizar en la actividad de cada etapa, los diagramas de descripción de la interacción enfatizan en cuál o cuáles elementos están implicados en la realización de esa actividad.

Diagramas de secuencia

Los diagramas de secuencia son una representación temporal de los objetos y sus interacciones; se centran en el tipo y orden de los mensajes que se pasan entre elementos durante la ejecución y son el tipo más común de diagrama de interacción y son muy intuitivos.

Cristian Pilaparia 1095211-2202112345

Diagramas de máquinas de estados

Estos diagramas capturan las transiciones internas de los estados de un elemento. El elemento puede ser tan pequeño como una única clase o tan grande como el sistema completo. Los diagramas de estado se utilizan frecuentemente para modelar los sistemas embebidos y especificaciones o implementaciones de protocolos.

Diagramas de tiempo

Son un tipo de diagrama de interacción que enfatiza especificaciones detalladas de tiempo en los mensajes. Se utilizan, por lo general, para modelar sistemas en tiempo real como sistemas de comunicación por satélites.

Diagramas de casos de uso

Estos diagramas representan las funciones del sistema desde el punto de vista del usuario, o lo que es igual, se utilizan para capturar los requerimientos funcionales de un sistema. Proporcionan una vista de implementación independiente de lo que hace un sistema y permiten al modelador centrarse en las necesidades del usuario en lugar de en detalles de realización.

UML tiene un amplio vocabulario para capturar el comportamiento y los flujos de procesos. Los diagramas de actividad y los de estado se pueden utilizar para capturar procesos de negocio que implican a personas, grupos internos o incluso organizaciones completas. UML 2.5.1 tiene una notación que ayuda considerablemente a modelar fronteras geográficas, responsabilidades del trabajador, transacciones complejas, etcétera.

Nota

UML es un lenguaje con una sintaxis y una semántica (vocabulario y reglas) que permiten una comunicación.

Es importante considerar que UML *no es* un proceso de software. Esto significa que se utilizará UML dentro de un proceso de software pero está concebido, claramente, para ser parte de un enfoque de desarrollo iterativo.

Desarrollo de software orientado a objetos con UML

UML tiene sus orígenes en la orientación a objetos. Por esta razón se puede definir UML como: "un lenguaje de modelado orientado a objetos para desarrollo de sistemas de software modernos". Al ser un lenguaje de modelado orientado a objetos, todos los elementos y diagramas en UML se basan en el paradigma orientado a objetos. El desarrollo orientado a objetos se centra en el mundo real y resuelve los problemas a través de la interpretación de "objetos" que representan los elementos tangibles de un sistema. La idea fundamental de UML es modelar software y sistemas como colecciones de objetos que interactúan entre sí. Esta idea se adapta muy bien al desarrollo de software y a los lenguajes orientados a objetos, así como a numerosas aplicaciones, como los procesos de negocios.

15.8. Bloques de construcción (componentes) de UML 2.5.1

Los bloques de construcción, elementos o cosas de UML 2.5 se dividen en cuatro categorías:

- Elementos estructurales.
- Elementos de comportamiento.
- Elementos de agrupación.
- Elementos de notación.

Cristian Pilapanta 1095211-2202112345

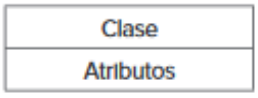
1095211220211234S

15.8.1. Elementos estructurales

Los elementos estructurales definen la parte estática del modelo y representan elementos conceptuales y físicos. La descripción de los elementos estructurales de UML más importantes son:

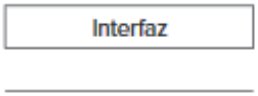
Clase

Una clase representa un conjunto de objetos que tienen responsabilidades similares.



Interfaz

Una interfaz define un conjunto de operaciones que especifican la responsabilidad de una clase.



Colaboración

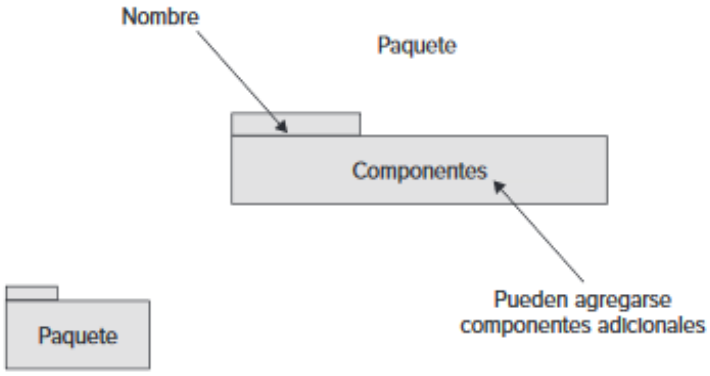
La colaboración define la interacción entre elementos.

15.8.3. Elementos de agrupación

Los elementos de agrupación se pueden definir como un mecanismo que agrupa los elementos de un modelo. Existe una sola cosa de agrupamiento disponible:

Paquete

El paquete es el elemento que permite agrupar elementos estructurales y de comportamiento.



Elementos de notación (notas)

Los elementos notacionales se definen como un mecanismo para capturar comentarios, descripciones y comentarios de los elementos del modelo UML. La nota es el único elemento notacional disponible.

Nota

Una nota se utiliza para realizar comentarios, restricciones, etc., de un elemento UML. Las notas añaden información a los diagramas del usuario para capturar cualquier cosa en su diagrama. Las notas se utilizan para expresar información adicional. Algunas herramientas permiten embeber enlaces URL en notas, proporcionando un medio para navegar de un diagrama al siguiente, documentos HTML.

15.9. Especificaciones de UML

El lenguaje unificado de modelado (**UML**, *The Unified Modeling Language*™), es la especificación más utilizada de OMG y el modo en que se usan los modelos del mundo, no sólo la estructura de la aplicación, comportamiento y arquitectura, sino también los procesos de negocio y estructura de datos. La documentación completa de UML se muestra en el sitio `www.uml.org` y en ella puede encontrar manuales, tutoriales, normas de certificaciones... para iniciarse en modelado, así como los beneficios de modelado en el desarrollo de sus aplicaciones.

La especificación actual oficial y las especificaciones de UML 1), así como los perfiles UML y especificaciones relacionadas 2) pueden descargarse en:

1. `www.omg.org/spec`
2. `www.omg.org/spec/UML/About-UML`
3. `www.omg.org/spec/UML/#Spec-versions-formal`

Si desea profundizar en UML le recomendamos que visite el sitio `www.omg.org` y consulte la bibliografía que se incluye en este libro. Por otra parte, los modelos UML tienen al menos dos dimensiones: una dimensión gráfica

Cristian Pilaparita 1095211-2202112345

para visualizar el modelo usando diagramas e íconos (notaciones), y otra dimensión con texto, que describe las especificaciones de distintos elementos de modelado.

La especificación o lenguaje OCL (Object Constraint Language) define un lenguaje para escritura de restricciones y expresiones de los elementos del modelo. La última versión de OCL recogida en el sitio web omg.org es la versión 2.4.

15.10. Historia de UML

En la primera mitad de la década de 1990, los lenguajes de modelado que imperaban eran: OMT-2 (Object Modeling Technique), creado por James Rumbaugh; OOSE (Object Oriented Software Engineering), creado por Ivan Jacobson y Booch'93,[2] cuyo creador era Grady Booch. Existían otros métodos que fueron muy utilizados: Coad/Yourdon y Fusion, entre otros. Fusion de Coleman constituyó un primer intento de unificación de métodos, pero al no contar con el apoyo de los restantes métodos, pese a ser apoyado por Hewlett-Packard en algún momento, no llegó a triunfar.

Grady Booch, propietario de Rational llamó a James Rumbaugh y formaron equipo en una nueva Rational Corporation con el objetivo de fusionar sus dos métodos. En octubre de 1994, Grady Booch y James Rumbaugh comenzaron a trabajar en la unificación de ambos métodos produciendo una versión en borrador llamada 0.8 y de nombre *Unified Method*.[3] Posteriormente, a partir de 1995, Jacobson se unió al tándem y formaron un equipo que se llegó a conocer como "los tres amigos".

El primer fruto de su trabajo colectivo se lanzó en enero de 1997 y fue presentado como versión 1.0 de UML. La gran ventaja de UML es que fue recogiendo aportaciones de los grandes gurús de objetos: David Hasel con sus diagramas de estado; partes de la notación de Fusion, el criterio de responsabilidad colaboración y los diagramas de Rebeca Wirfs-Brock, y el trabajo de patrones y documentación de Gamma-Helm-Johnson-Ulissides.

En 1997, OMG aceptó UML como estándar (versión 1.1: `www.omg.org/spec/UML/1.1`) y nació el primer lenguaje de modelado visual orientado a objetos como estándar abierto de la industria. Desde entonces han desaparecido, prácticamente, todas las demás metodologías y UML se ha convertido en el estándar de la industria del software y de muchas otras industrias que requieren el uso de modelos. En 1998, OMG lanzó dos revisiones más, 1.2 y 1.3. En el año 2000 se presentó UML 1.4 con la importante aportación de la semántica de acción, que describe el comportamiento de un conjunto de acciones permitidas que pueden implementarse mediante lenguajes de acción. La versión 1.5 siguió a las ya citadas.

En 2005 se lanzó la versión 2.0 con su especificación completa y se convirtió en un lenguaje de modelado muy maduro. **UML 2** ha incorporado numerosas mejoras a **UML 1**. Los cambios más importantes se han realizado en el *metamodelo* (generador de modelos), conservando los principios fundamentales y evolutivos de las últimas versiones. Los diseñadores de UML 2.0 (ya no sólo los tres creadores originales, sino una inmensa pléyade que ha contribuido con infinitas aportaciones) han tenido mucho cuidado en asegurar que fuera totalmente compatible con las versiones anteriores para que los usuarios de éstas no tuvieran problemas de adaptación.

UML ha seguido evolucionando: versión 2.1 en abril de 2006; 2.2 en febrero de 2009; en mayo de 2010, la versión 2.3; y la versión 2.4.1 en agosto de 2011. La última versión presentada ha sido **UML 2.5.1** en diciembre de 2017. En la página oficial de OMG `www.omg.org` encontrará la información más actualizada, así como especificaciones y manuales de todas las versiones disponibles.

Línea de tiempo (cronología, *timeline*)

- 1997 – UML 1.1
- 1997 – OMG UML 1.1 (a partir de esta fecha todas las versiones son OMG UML)
- 1998 – UML 1.2
- 1999 – UML 1.3
- 2001 – UML 1.4.2 (norma ISO/IEC 19501)
- 2003 – UML 1.5
- 2005 – UML 2.0

[2] Los libros originales de OMT y Booch'93 los tradujo un equipo de profesores universitarios dirigidos por el profesor Joyanes.
[3] Todavía conservo el borrador de una versión, en papel, que fue presentada por James Rumbaugh, entre otros lugares, en un hotel en Madrid, invitado por su editorial Prentice-Hall (nota de Luis Joyanes).

Cristian Pilapaña 1095211-2202112345

- 2006 – UML 2.1
- 2007 – UML 2.1.1/2.1.2
- 2009 – UML 2.2
- 2010 – UML 2.3
- 2011 – UML 2.4.1 (aprobado en julio de 2011)
- 2017 – UML 2.5.1 (aprobado en diciembre de 2017)

CONCEPTOS CLAVE

- Abstracción
- ADT
- Atributos
- Clase
- Clase base
- Clase derivada
- Comportamiento
- Comunicación entre objetos
- Encapsulamiento
- Función miembro
- Herencia
- Herencia múltiple
- Herencia simple
- Instancia
- Ligadura
- Mensaje
- Método
- Objeto
- Objeto compuesto
- Operaciones
- Polimorfismo
- Reutilización
- Reusabilidad
- Sobrecarga
- TDA 1 TAD (ADT)
- Tipo Abstracto de Datos
- Variable de instancia

Cristian Pilaparita 1095211-2202112346

RESUMEN

El *tipo abstracto de datos* se implementa a través de *clases*. Una **clase** es un conjunto de objetos que constituyen instancias de ella, cada una de las cuales tiene la misma estructura y comportamiento. Una clase tiene un nombre, una colección de operaciones para manipular sus instancias y una representación. Las operaciones que manipulan las instancias de una clase se llaman *métodos*. El estado o representación de una instancia se almacena en variables de instancia. Estos métodos se invocan mediante el envío de *mensajes* a instancias. El envío de mensajes a objetos (instancias) es similar a la llamada a procedimientos en lenguajes de programación tradicionales.

El *polimorfismo* permite desarrollar sistemas en los que objetos diferentes pueden responder de modo distinto al mismo mensaje.

La programación orientada a objetos incorpora estos seis componentes importantes:

- Objetos.
- Clases.
- Métodos.
- Mensajes.
- Herencia.
- Polimorfismo.

Un objeto se compone de datos y funciones que operan sobre esos objetos.

La técnica de situar datos dentro de objetos, de modo que no se puede acceder directamente a los datos, se llama *ocultación de la información.*

Una *clase* es una descripción de un conjunto de objetos. Una *instancia* es una variable de tipo objeto y *un objeto es una instancia de una clase.*

La *herencia* es la propiedad que permite a un objeto pasar sus propiedades a otro objeto, o dicho de otro modo, un objeto puede heredar de otro objeto.

Los objetos se comunican entre sí pasando *mensajes.*

La clase padre o ascendiente se denomina *clase base* y las clases descendientes, *clases derivadas.*

La *reutilización de software* es una de las propiedades más importantes que presenta la programación orientada a objetos.

El *polimorfismo* es la propiedad por la cual un mismo mensaje puede actuar de diferente modo cuando actúa sobre objetos distintos ligados por la propiedad de la herencia.

UML, Lenguaje Unificado de Modelado, es utilizado en el desarrollo de sistemas.

UML tiene dos categorías importantes de diagramas: *diagramas estructurales* (de estructura) y de *comportamiento*. Las construcciones contenidas en cada uno de los diagramas UML se describen al lado de cada diagrama:

- Diagrama de actividad – actividades.
- Diagrama de clases – clasificadores estructurados.
- Diagrama de comunicación – interacciones.
- Diagrama de componentes – clasificadores estructurados.
- Diagrama compuesto de estructura – clasificadores estructurados.
- Diagrama de despliegue – despliegues.
- Diagrama de interacción global – interacciones.
- Diagrama de objetos – clasificación.
- Diagrama de paquetes – paquetes.
- Diagrama de perfiles – paquetes.
- Diagrama de máquina de estados – máquina de estados.
- Diagrama de secuencias – interacciones.
- Diagramas de tiempo – interacciones.
- Diagramas de casos de uso – casos de uso.

ANEXO: TERMINOLOGÍA DE ORIENTACIÓN A OBJETOS

Los lenguajes de programación orientada a objetos utilizados en la actualidad son numerosos y aunque la mayoría siguen criterios de terminología universales puede haber algunas diferencias relativas a su consideración de *puros* (Smalltalk, Eiffel...) e *híbridos* (Object Pascal, VB.NET, C++, Java, C#...). La tabla 15.1 sintetiza la terminología utilizada en los manuales de programación de cada respectivo lenguaje.

Tabla 15.1. Terminología de orientación a objetos en diferentes lenguajes de programación.

Concepto	Object Pascal	VB.NET	C++	Java	C#	Smalltalk	Eiffel
Objeto	Objeto	Objeto	Objeto	Objeto	Objeto	Objeto	Objeto
Clase	Tipo-Objeto	Clase	Clase	Clase	Clase	Clase	Clase
Método (*function* miembro)	Método	Método	Función Miembro	Método	Método	Método	Rutina
Mensaje	Mensaje	Mensaje	Mensaje	Mensaje	Mensaje	Mensaje	Aplicación
Herencia	Herencia	Herencia	Herencia	Herencia	Herencia	Herencia	Herencia
Superclase		Clase Base	Clase Base	Superclase	Clase Base	Superclase	Ascendiente
Subclase	Descendiente	Clase Derivada	Clase Derivada	Subclase	Clase Derivada	Subclase	Descendiente

EJERCICIOS

15.1. Describa y justifique los objetos que obtiene de cada uno de estos casos:

a) Los habitantes de Europa y sus direcciones de correo.
b) Los empleados de una empresa y sus claves de acceso a sistemas de seguridad.

15.2. ¿Cuáles serían los objetos que deben considerarse en los siguientes sistemas?

a) Un programa para maquetar una revista.
b) Un contestador telefónico.
c) Un sistema de control de ascensores.
d) Un sistema de suscripción a una revista.

Cristian Pilapanta 1095211-2202112346

15.3. Definir los siguientes términos:

a) Clase.
b) Objeto.
c) Sección de declaración.
d) Sección de implementación.
e) Variable de instancia.
f) Función miembro.
g) Miembro dato.
h) Constructor.
i) Instancia de una clase.
j) Métodos o servicios.
k) Sobrecarga.
l) Interfaz.

15.4. Deducir los objetos necesarios para diseñar un programa de computadora que permita jugar a diferentes juegos de cartas.

15.5. Determinar los atributos y operaciones que pueden ser de interés para los siguientes objetos, partiendo de la base que van a ser elementos de un almacén de regalos: un libro, un disco, una grabadora de video, una cinta de video, un televisor, una radio, un tostadora de pan, una cadena de música, una calculadora y un teléfono celular (móvil).

15.6. Crear una clase que describa un rectángulo que se pueda visualizar en la pantalla de la computadora, cambiar de tamaño, modificar su color de fondo y los colores de los lados.

15.7. Representar una clase ascensor (elevador) que tenga las funciones usuales de subir, bajar, parar entre niveles (pisos), alarma, sobrecarga y en cada nivel (piso) botones de llamada para subir o bajar.

15.8. Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo `Figura`.

15.9. Construir una clase `Persona` con las funciones miembro y atributos que crea oportunos.

15.10. Construir una clase llamada `Luz` que simule una luz de tráfico. El atributo color de la clase debe cambiar de `Verde` a `Amarillo` y a `Rojo` y de nuevo regresar a `Verde` mediante la función `Cambio`. Cuando un objeto `Luz` se crea su color inicial será `Rojo`.

15.11. Construir una definición de clase que se pueda utilizar para representar un `empleado` de una compañía. Cada empleado se define por un número entero ID, un salario y el número máximo de horas de trabajo por semana. Los servicios que debe proporcionar la clase, al menos deben permitir introducir datos de un nuevo empleado, visualizar los datos existentes de un nuevo empleado y capacidad para procesar las operaciones necesarias para dar de alta y de baja en la seguridad social y en los seguros que tenga contratados la compañía.

Cristian Pilaparda 1095211-2202112346

CAPÍTULO 16

Diseño de clases y objetos: representaciones gráficas en UML

16.1. Diseño y representación gráfica de objetos en UML
16.2. Diseño y representación gráfica de clases en UML
16.3. Declaración de objetos de clases
16.4. Constructores
16.5. Destructores
16.6. Implementación de clases en C++
16.7. Recolección de basura
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS
LECTURAS RECOMENDADAS

Cristian Pilaparia 1095211-2202112346

1095211220211234

INTRODUCCIÓN

Hasta ahora hemos aprendido el concepto de estructuras, con lo que se ha visto un medio para agrupar datos. También se han examinado funciones que sirven para realizar acciones determinadas a las que se les asigna un nombre. En este capítulo se tratarán las clases, un nuevo tipo de dato cuyas variables serán objetos. Una **clase** es un tipo de dato que contiene código (funciones) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, tal como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En este capítulo se aprenderá a crear (definir y especificar) y utilizar clases individuales y en el capítulo 17 se verá cómo definir y utilizar jerarquías y otras relaciones entre clases.

El diagrama de clases es uno de los tipos de diagramas fundamentales en UML. Se utiliza para capturar las relaciones estáticas de su software; en otras palabras, cómo poner o relacionar cosas juntas. Cuando se escribe software se está constantemente tomando decisiones de diseño: qué clases hacen referencia a otras clases, qué clases "poseen" alguna otra clase, etcétera. Los diagramas de clase proporcionan un medio para capturar la estructura física de un sistema.

El paradigma orientado a objetos nació en 1969 de la mano del doctor noruego Kristin Nygaard que intentando escribir un programa de computadora que describiera el movimiento de los barcos a través de un fiordo, descubrió que era muy difícil simular las mareas, los movimientos de los barcos y las formas de la línea de la costa con los métodos de programación existentes en ese momento. Descubrió que los elementos del entorno que trataba de modelar —barcos, mareas y línea de la costa de los fiordos— y las acciones que cada elemento podía ejecutar, constituían unas relaciones que eran más fáciles de manejar.

Las tecnologías orientadas a objetos han evolucionado mucho, pero mantienen la razón de ser del paradigma: combinación de la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas, son los elementos clave sobre los que se articula la orientación a objetos.

16.1. Diseño y representación gráfica de objetos en UML

Un objeto es la instancia de una clase, por ejemplo, el señor Mackoy es un objeto de la clase Persona. Un objeto es simplemente una colección de información relacionada y funcionalidad; puede ser algo que tenga una manifestación o correspondencia en el mundo real (tal como un objeto empleado), algo que tenga algún significado virtual (tal como una ventana en la pantalla) o alguna abstracción adecuada dentro de un programa (una lista de trabajos a realizar, por ejemplo). Es una entidad atómica formada por la unión del estado y del comportamiento; proporciona una relación de encapsulamiento que asegura una fuerte cohesión interna y un débil acoplamiento con el exterior. Además, revela su verdadero rol y la responsabilidad cuando al enviar mensajes se convierte en parte de un escenario de comunicaciones; contiene su propio estado interno y un comportamiento accesible a otros objetos.

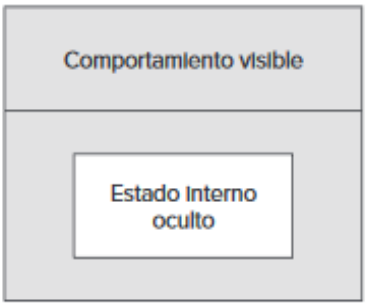


Figura 16.1. Objeto: estado y comportamiento.

El mundo en el que vivimos se compone de objetos tangibles de todo tipo; el tamaño de éstos es variable, pueden ser pequeños como un grano de arena o grandes como una montaña o un buque de recreo. Nuestra idea intuitiva de objeto viene directamente relacionada con el concepto de masa, es decir, la propiedad que caracteriza la cantidad de materia dentro de un determinado cuerpo. Sin embargo, es posible definir otros objetos que no tengan ninguna masa, tal como una cuenta corriente, una póliza de seguros, una ecuación matemática o los datos personales de un alumno de una universidad. Estos objetos corresponden a conceptos, en lugar de a entidades físicas.

Se puede ir más lejos y extender la idea de objeto haciendo que pertenezcan a "mundos virtuales" (asociados con la red internet, por ejemplo) con la finalidad de crear comunidades de personas que no estén localizadas en la misma área geográfica. Objetos de software definen una representación abstracta de las entidades del mundo real con el objetivo de controlarlo o simularlo. Los objetos de software pueden ir desde listas enlazadas, árboles o grafos hasta archivos completos o interfaces gráficas de usuario.

En síntesis, un objeto se compone de datos que lo describen y de operaciones que se pueden ejecutar sobre el mismo. La información almacenada en un objeto empleado, por ejemplo, puede ser de identificación (nombre, dirección, edad, titulación), laboral (título del trabajo, salario, antigüedad), etcétera. Las operaciones realizadas pueden incluir la creación del sueldo del empleado o la promoción de este último.

Al igual que los objetos del mundo real que nacen, viven y mueren, los objetos del mundo del software tienen una representación similar conocida como su ciclo de vida.

Nota

Un objeto es algo que encapsula información y comportamiento. Es un término que representa una cosa concreta o del mundo real.

EJEMPLO DE OBJETOS

- Vuelo 6520 de Iberia (Santo Domingo-Madrid con escala en San Juan de Puerto Rico).
- Casa núm. 31 de la avenida Andalucía, en Carchelejo (Jaén).
- Flor roja en el balcón de la terraza del profesor Mackoy.

Cristian Pilapaña 1095211-2202112346

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos; tienen propiedades específicas, tales como posición, tamaño, color, forma, textura, etcétera, que definen su estado; además, cuentan con ciertos comportamientos que los hacen diferentes de otros objetos.

Booch[1] define un *objeto* como "algo que tiene un estado, un comportamiento y una identidad". Imaginemos la máquina de una fábrica; el *estado* de la *máquina* puede estar *funcionando/parando* ("on/off"), su potencia, velocidad máxima, velocidad actual, temperatura, etcétera. Su *comportamiento* puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, condiciones de señal de error o cambiar la velocidad. Su *identidad* se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar en el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En el diseño de un programa orientado a objetos se crea una abstracción (un modelo simplificado) de la máquina basada en las propiedades y comportamiento que son útiles en el tiempo.

[Martin/Odell] definen un objeto como "cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos". Para realizar esa actividad se añaden a cada objeto de la clase los propios datos asociados con sus propias funciones miembro que pertenecen a la clase.

Cualquier programa orientado a objetos puede manejar muchos objetos. Por ejemplo, un programa que maneja el inventario de un almacén de ventas al por menor utiliza un objeto de cada producto manipulado en el almacén; el programa manipula los mismos datos de cada objeto, incluyendo el número de producto, la descripción, precio, número de artículos del stock y el momento de nuevos pedidos.

Cada objeto conoce también cómo ejecutar acciones con sus propios datos. El objeto producto del programa de inventario, por ejemplo, conoce cómo crearse a sí mismo y establecer los valores iniciales de todos sus datos, cómo modificar sus datos y cómo evaluar si hay artículos suficientes en el *stock* para cumplir una petición de compra. En esencia, la cosa más importante de un objeto es reconocer que consta de datos, y las acciones que puede ejecutar.

Un objeto de un programa de computadora es algo que no se puede tocar. Cuando un programa se ejecuta, la mayoría existe en la memoria principal. Los objetos se crean por un programa para su uso mientras éste se está ejecutando. A menos que se guarden los datos de un objeto en un disco, el objeto se pierde cuando el programa termina (este objeto se llama *transitorio* para diferenciarlo del objeto *permanente* que se mantiene después de la terminación del programa).

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones; éste incluye un identificador que contiene la acción y los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con éste mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico tal como una máquina de fax tiene una interfaz de usuario bien definida; por ejemplo, esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón "enviar". El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles. De hecho, la apertura de la máquina durante el período de garantía puede anularla.

16.1.1. Representación gráfica en UML

Un objeto es una instancia de una clase; es decir, se pueden tener varias instancias de una clase llamada `Carro (Coche)`. Un carro rojo de dos puertas, un carro azul de cuatro puertas y un carro verde todo terreno de cinco puertas. Cada instancia de `Carro` es un objeto al que se puede dar un nombre o dejarlo anónimo y se representa en los diagramas de objetos. Normalmente se muestra el nombre del objeto seguido por el símbolo dos puntos y el nombre de la clase o su tipo. Tanto el nombre del objeto como el nombre de la clase se subrayan. En UML un objeto se representa por un rectángulo en cuyo interior se escribe el nombre del objeto subrayado.

El diagrama de representación tiene tres modelos (figura 16.2).

En la figura 16.3 se muestra un diagrama que representa diferentes clientes de un banco y las cuentas asociadas con cada uno de estos clientes. Las líneas que conectan estos objetos representan los enlaces que existen entre un cliente determinado y sus cuentas. El diagrama muestra también un rectángulo con un doblete en la esquina superior

[1] Booch, Grady: *Análisis y diseño orientado a objetos con aplicaciones*, Madrid, Díaz de Santos/Addison-Wesley, 1995.

Cristian Pilapanta 1095211-2202112346

Figura 16.2. Diferentes objetos: `Seat` y `BMW` de la clase `Carro`, `milavadora` de la clase `Lavadora`.

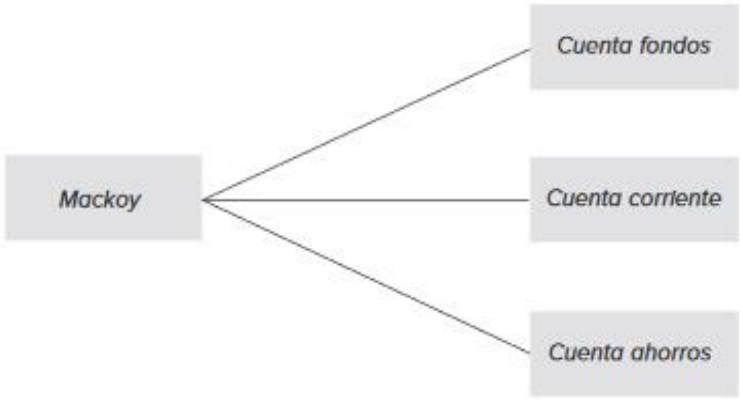


Figura 16.3. Enlaces entre objetos de las clases `Cliente` y `Cuenta`.

Cristian Pilaparia 1095211-2202112346

derecha; este diagrama representa un comentario (una nota, un texto de información libre concebida con propósito de clarificación de la figura y de facilitar la comprensión del diagrama); las líneas punteadas implementan la conexión de cualquier elemento del modelo a una nota descriptiva o de comentario.

A veces es difícil encontrar un nombre para cada objeto, por esta razón suele utilizarse con mayor frecuencia un nombre genérico en lugar de un nombre individual. Esta característica permite nombrar los objetos con términos genéricos y evitar abreviaturas de nombres o letras, tal como se hacía antiguamente, `a`, `b` o `c`.

El siguiente diagrama (figura 16.4) muestra estudiantes y profesores. La ausencia de cualquier texto precedente delante de los dos puntos significa que estamos hablando de tipos de objetos genéricos o anónimos de tipos `Estudiante` y `Profesor`.

Figura 16.4. Diferentes representaciones de objetos (ejemplo: `Profesor` y `Estudiante`).

16.1.2. Características de los objetos

Todos los objetos tienen tres características o propiedades fundamentales que sirven para definir a un objeto de modo inequívoco: *un estado*, *un comportamiento* y *una identidad*.

Objeto = Estado + Comportamiento + Identidad

Un objeto debe tener todas o alguna de las propiedades anteriores. Puede existir un objeto sin estado o sin comportamiento, pero siempre deberá tener una identidad.

Booch define un objeto a partir de su experiencia y la de sus colegas como:

> Un objeto es una entidad que tiene estado, comportamiento e identidad. La estructura y comportamiento de objetos similares se definen en sus clases comunes. Los términos *instancia* y *objeto* son intercambiables (Booch *et al.*, 2007).

16.1.3. Estado

El estado agrupa los valores de todos los **atributos** de un objeto en un momento dado, un atributo es una pieza de información que califica el objeto contenedor. Cada atributo puede tomar un valor en un dominio de definición dado. *En algún momento, el estado de un objeto se corresponde con una selección determinada de valores a partir de valores posibles de los diversos atributos.* En esencia, un atributo es una propiedad o característica de una clase y describe un rango de valores que la propiedad podrá contener en los objetos de la clase. Una clase podría contener ninguno o varios atributos.

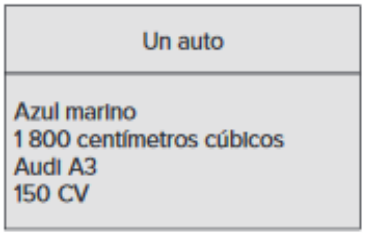


Figura 16.5. Un objeto con sus atributos.

Cristian Pilapanta 1095211-2202112346

1095211220211234 6

> El estado de un objeto abarca todas las propiedades (normalmente estáticas) del objeto más los valores actuales (normalmente dinámicos) de cada una de estas propiedades (Booch *et al.*, 2007).

Los atributos de una clase son las partes de información que representan el estado de un objeto y constituyen las propiedades de una clase. Así, los detalles de la clase `Carro` son atributos: color, número de puertas, potencia, entre otros. Los atributos pueden ser tipos primitivos simples (enteros, reales...), compuestos (cadena, complejo...) o relaciones a otros objetos complejos.

Una clase puede tener cero o más atributos. Un nombre de un atributo puede ser cualquier conjunto de caracteres, pero dos atributos de la misma clase no pueden tener el mismo nombre. Un atributo se puede mostrar utilizando dos notaciones diferentes: *en línea* o *en relaciones entre clases*. Además, la notación está disponible para representar otras propiedades, tales como multiplicidad, unicidad u ordenación.

Por convenio, el nombre de un atributo puede ser de una o varias palabras unidas. Si el nombre es de una palabra se escribe en minúsculas; si es de más, las palabras se unen y cada una, excepto la primera, comienzan con letra mayúscula. La lista de atributos se sitúa en el compartimento o banda debajo del compartimento que contiene al nombre de la clase.

Nombre de objetos

```
miLavadora : Lavadora                    instancia con nombre de instancia
:Lavadora                                sin nombre (anónima)
```

UML proporciona la opción de indicar información adicional para los atributos. En la notación de la clase, se puede especificar un tipo para cada valor del atributo; se pueden incluir tipos tales como cadena (*string*), número de coma flotante, entero o *boolean* (y otros tipos enumerados). Para indicar un tipo, se utilizan dos puntos (:) para separar el nombre del atributo del tipo. Se puede indicar también un valor por defecto para un atributo.

nombre : tipo = valor_por_defecto

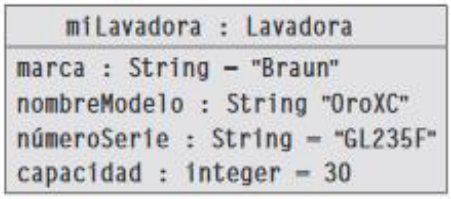


Figura 16.6. Atributos con valores por defecto.

Un atributo se representa con una sola palabra en minúsculas; por otro lado, si el nombre contiene más de una palabra, cada una será unida a la anterior y comenzará con una letra mayúscula, a excepción de la primera, que comenzará con minúscula. La lista de atributos se inicia en la segunda banda del ícono de la clase. Todo objeto de la clase tiene un valor específico en cada atributo. El nombre de un objeto se inicia con una letra minúscula y está precedido de dos puntos que a su vez están precedidos del nombre de la clase, y todo está subrayado.

El nombre `miComputadora:Computadora` es una instancia con nombre (un objeto), pero también es posible tener un objeto o instancia anónima y se representa tal como `:Computadora`.

EJEMPLOS DE OBJETOS

- El profesor Mariano.
- La ciudad de Toledo.
- La casa en Calle Real 25, Carchelejo (Jaén).
- El coche (carro) amarillo que está estacionado en la calle al lado de mi ventana.

Cada objeto encapsula una información y un comportamiento. Por ejemplo, en el objeto `vuelo IB 6170` la fecha de salida es el 16 de agosto de 2002; la hora, 10:30 de la mañana; el número de vuelo, 6170; la compañía de aviación, Iberia; la ciudad de partida, Santo Domingo y la ciudad destino es Madrid, con breve escala en San Juan de Puerto Rico. El objeto `Vuelo` también tiene un comportamiento. Se conocen los procedimientos de cómo añadir un pasajero al vuelo, quitarlo o determinar cuándo el vuelo está lleno; es decir, `añadir`, `quitar`, `estáLleno`. Aunque los valores de los atributos cambiarán con el tiempo (el vuelo IB 6520 tendrá una fecha de salida, el día siguiente, 17 de agosto), los atributos por sí mismos nunca cambiarán. El vuelo 6170 siempre tendrá una fecha de salida, una hora y una ciudad de salida; es decir, sus atributos son fijos.

En UML se pueden representar los tipos y valores de los atributos. Para indicar un tipo, utilice dos puntos (:) para separar el nombre del atributo de su tipo.

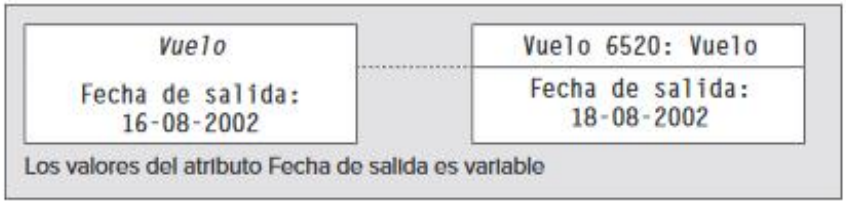


Figura 16.7. Un objeto con atributos, sus tipos así como sus valores predeterminados.

Cristian Pilaparita 1095211-2202112346

16.1.4. Múltiples instancias de un objeto

En un diagrama de clases se pueden representar diversas instancias de un objeto mediante íconos múltiples. Por ejemplo, si se necesita representar una lista de vuelos de Iberia en un diagrama de clases u objetos, en lugar de mostrar cada vuelo como un objeto independiente se puede utilizar un ícono con múltiples instancias para mostrar la lista de vuelos. La notación UML para representar instancias múltiples se muestra en la figura 16.8.

Figura 16.8. Instancias múltiples del objeto `Vuelo`.

Nota

Los atributos son los trozos de información contenidos en un objeto. Los valores de los atributos pueden cambiar durante la vida del objeto.

Cristian Pilapanta 1095211-2202112346

16.1.5. Evolución de un objeto

El estado de un objeto evoluciona con el tiempo. Por ejemplo, el objeto `Auto` tiene los atributos: `Marca`, `Color`, `Modelo`, `Capacidad` del depósito o tanque de la gasolina, `Potencia` (en caballos de vapor). Si el auto comienza un viaje, normalmente se llenará el depósito (el atributo `Capacidad` puede tomar, por ejemplo, el valor 50 "litros" o 12 galones), el `color` del auto, en principio no cambiará (`azul cielo`), la `potencia` tampoco cambiará (`150 Caballos, HP`). Es decir, hay atributos cuyo valor va variando, tal como la capacidad (ya que a medida que avance, disminuirá la cantidad que contiene el depósito), pero habrá otros que normalmente no cambiarán, como el color y la potencia del auto, o la marca y el modelo, inclusive el país en donde se ha construido.

El diagrama de la figura 16.9 representa la evolución de la clase `Auto` con un comentario explicativo de la disminución de la gasolina del depósito debido a los kilómetros recorridos.

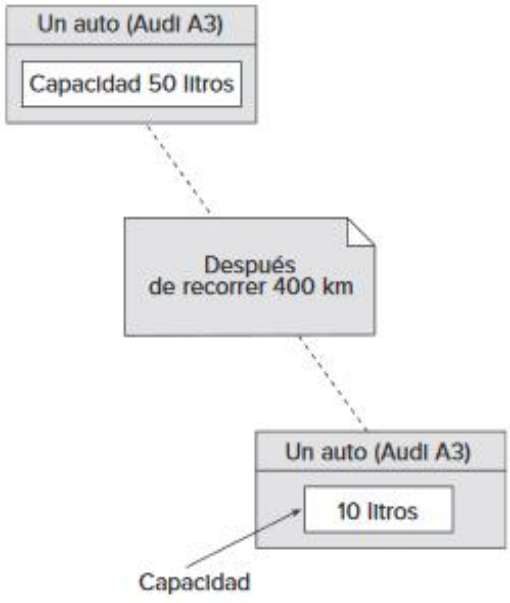


Figura 16.9. Evolución de una clase.

Los objetos de software encapsulan una parte del conocimiento del mundo en el que ellos evolucionan.

16.1.6. Comportamiento

El comportamiento es el conjunto de capacidades y aptitudes de un objeto y describe las acciones y reacciones del mismo. Cada componente del comportamiento individual de un objeto se denomina **operación**, y es algo que la clase puede realizar o que se puede hacer a una clase. Las operaciones de un objeto se disparan (activan) como resultado de un estímulo externo representado en la forma de un mensaje enviado a otro objeto.

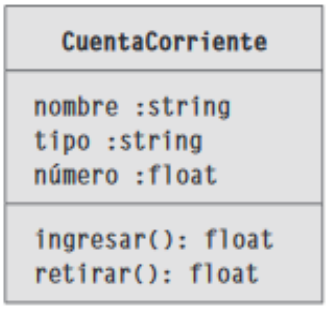
Las operaciones son las características de las clases que especifican el modo de invocar un comportamiento específico. Una operación de una clase describe qué hace pero no necesariamente cómo lo hace. Por ejemplo, una clase puede ofrecer una operación para dibujar un rectángulo en la pantalla, o bien contar el número de elementos seleccionados de una lista. UML hace una diferencia clara entre la especificación de cómo invocar un comportamiento (una operación) y la implementación real de ese comportamiento (método o función).

Las operaciones en UML se especifican en un diagrama de clase con una estructura compuesta por nombre, un par de paréntesis (vacíos o con la lista de parámetros que necesita la operación) y un tipo de retorno.

Sintaxis operaciones

1. ***nombre (parámetros) : tipo_retorno***
2. `nombre()`

Cristian Pilapanta 1095211-2202112346



1095211220211234 6

Figura 16.10. Clase `CuentaCorriente`.

Al igual que sucede con los nombres de los atributos, el nombre de una operación se pone en minúsculas si es una palabra; en el caso de que el nombre conste de más de una palabra se unen ambas y comienzan todas las palabras reservadas después de la primera con una letra mayúscula. Así, la clase `Lavadora`, por ejemplo, puede tener las siguientes operaciones:

```
aceptarRopa (r: String)
aceptarDetergente (d: String)
darBotónPrender: Boolean
darBotónApagar: Boolean
```

Comportamiento es el modo en que un objeto actúa y reacciona, en términos de sus cambios de estado y paso de mensajes (Booch *et al.*, 2007).

En la figura 16.11, se disparan una serie de interacciones dependiendo del contenido del mensaje.

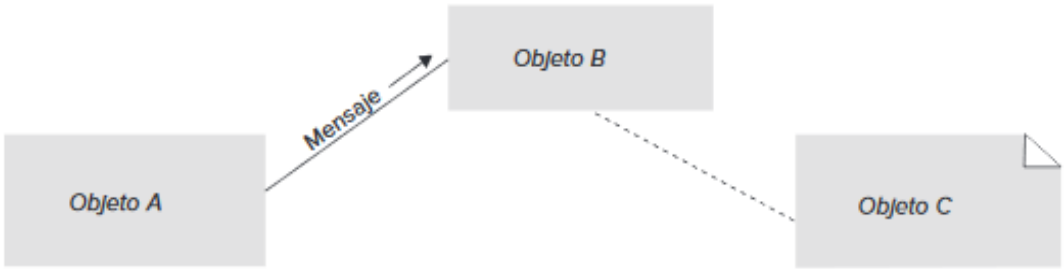


Figura 16.11. Mensaje entre objetos.

Las interacciones entre objetos se representan utilizando diagramas en donde los objetos que interactúan se unen a los restantes vía líneas continuas denominadas **enlaces**. La existencia de un enlace indica que un objeto conoce o ve a otro objeto. Los mensajes navegan junto a los enlaces, por lo general en ambas direcciones.

Ejemplo: El objeto `A` envía un mensaje `Almorzar` al objeto `B` y éste envía un mensaje `EcharLaSiesta` al objeto `C`. Las operaciones que se realizan mediante la comunicación de mensajes presuponen que el objeto `B` tiene la capacidad de almorzar y que el objeto `C` es capaz de irse a echar la siesta (figura 16.12).

Figura 16.12. Envío de mensajes.

El estado y el comportamiento están enlazados; realmente, el comportamiento en un momento dado depende del estado actual y el estado puede ser modificado por el comportamiento. Sólo es posible aterrizar un avión si está volando, de modo que el comportamiento `aterrizar` sólo es válido si la información `enVuelo` es verdadero. Después de aterrizar la información `enVuelo` se vuelve `falsa` y la operación `aterrizar` ya no tiene sentido; en este caso tendría sentido `despegar`, ya que la información del atributo `enVuelo` es falsa, cuando el avión está en tierra pendiente de despegar. El diagrama de colaboración de clases ilustra la conexión entre el estado y el comportamiento de los objetos de las clases. En el caso del objeto `Vuelo 6520`, las operaciones del objeto `Vuelo` pueden ser añadir o quitar un pasajero y verificar si el vuelo está lleno.

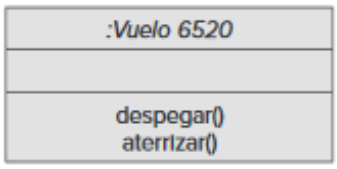


Figura 16.13. Objeto `Vuelo 6520`.

Definición

El comportamiento de un objeto es el conjunto de sus operaciones.

Regla

De igual manera que el nombre de un atributo, el nombre de una operación se escribe con minúsculas si consta de una sola palabra. En caso de constar de más de una palabra, éstas se unen e inicia cada una con mayúscula excepto la primera. La lista de operaciones se inicia en la tercera banda del ícono de la clase, justo debajo de la línea que separa las operaciones de los atributos.

Cristian Pilapaña 1095211-2202112346

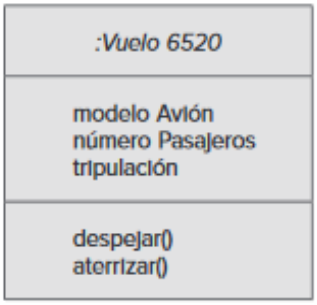


Figura 16.14. Representación gráfica de una clase con estado y comportamiento.

16.1.7. Identidad

La **identidad** es la propiedad que diferencia un objeto de otro objeto similar. En esencia, la identidad de un objeto caracteriza su propia existencia y hace posible distinguir cualquier objeto sin ambigüedad, independientemente de su estado. Esto permite, entre otras cosas, la diferenciación de dos objetos que tengan los atributos idénticos.

La identidad no se representa específicamente en la fase de modelado de un problema. Cada objeto tiene implícitamente una identidad. Durante la fase de implementación, la identidad se crea normalmente utilizando un identificador que viene naturalmente del dominio del problema. Nuestros autos tienen un número de placa, nuestros teléfonos celulares tienen un número a donde podemos ser llamados y nosotros mismos podemos ser identificados por el número del pasaporte, el de seguridad social o el ID correspondiente. El tipo de identificador, denominado también "clave natural", se puede añadir a los estados del objeto a fin de diferenciarlos. Sin embargo, sólo es un artefacto de implementación, de modo que el concepto de identidad permanece independiente del concepto de estado.

Cristian Pilapaña 1095211-2202112346

16.1.8. Los mensajes

El mensaje es el fundamento de una relación de comunicación que enlaza dinámicamente los objetos que fueron separados en el proceso de descomposición de un módulo. En la práctica, un **mensaje** es una comunicación entre objetos en los que un objeto (el cliente) solicita al otro objeto (el proveedor o servidor) hacer o ejecutar alguna acción.

Figura 16.15. Comunicación entre objetos.

También se puede mostrar en UML mediante un diagrama de secuencia (figura 16.16).

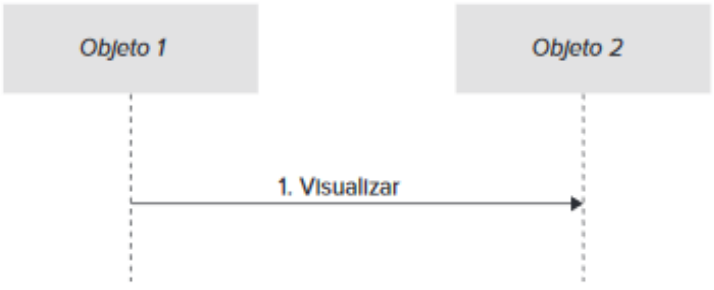


Figura 16.16. Diagrama de secuencia.

El mensaje puede ser reflexivo: un objeto se envía un mensaje a sí mismo:

Figura 16.17. Mensaje reflexivo.

La noción de un mensaje es un concepto abstracto que se puede implementar de varias formas, tales como una llamada a una función, un evento o suceso directo, una interrupción o una búsqueda dinámica, entre otros. En realidad un mensaje combina flujos de control y flujos de datos en una única entidad. Las flechas simples indican el flujo de control y las que cuentan con un pequeño círculo en el origen son flujos de datos.

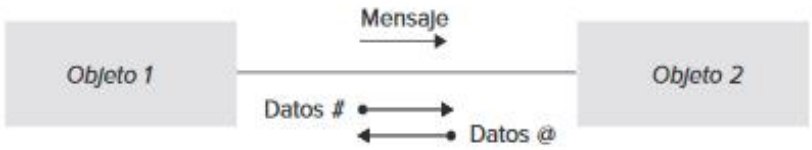


Figura 16.18. Representación gráfica de un mensaje con flujos de control y de datos.

Cristian Pilaparita 1095211-2202112346

Tipos de mensajes

Existen diferentes categorías de mensajes:

- *Constructores* (crean objetos).
- *Destructores* (destruyen objetos).
- *Selectores* (devuelven todo o parte del estado de un objeto).
- *Modificadores* (cambian todo o parte del estado de un objeto).
- *Iteradores* (visitan el estado de un objeto o el contenido de una estructura de datos que incluyen varios objetos).

EJEMPLO

Esquema de una clase con métodos o funciones correspondientes a los tipos de mensajes.

```
clase VueloAvión
   público
   // constructores
   ...
   // destructores
   ...
   // selectores
   ...
   // modificadores
   ...
   // iteradores
```

```
   ...
privado
   // atributos del vuelo
fin_clase
```

16.1.9. Responsabilidad y restricciones

El ícono de la clase permite especificar otro tipo de información sobre la misma: responsabilidad. La *responsabilidad* es un contrato o una obligación; es una descripción de lo que debe hacer la clase. Al crear una clase se está expresando que todos sus objetos tienen el mismo tipo de estado y el mismo tipo de comportamiento. A un nivel más abstracto, estos atributos y operaciones son simplemente las características por medio de las cuales se llevan a cabo las responsabilidades de la clase (Booch 06). Así una clase `Lavadora` tiene la responsabilidad de: "recibir ropa sucia como entrada y producir ropa limpia como salida". Una clase `Pared` de una casa es responsable de conocer la altura, anchura, grosor y color de la pared; una clase `SensorDeTemperatura` es responsable de medir la temperatura del carro y disparar la alarma (el piloto rojo de peligro) si esta temperatura alcanza un valor determinado de riesgo.

Las responsabilidades de la clase se pueden escribir en la zona inferior, debajo del área que contiene la lista de operaciones. En el ícono de la clase se debe incluir la suficiente información para describirla de un modo no ambiguo. La descripción de las responsabilidades de la clase es un modo informal de eliminar su ambigüedad.

Una *restricción* es un modo más formal de eliminar la ambigüedad. La regla que describe la restricción es encerrar entre llaves el texto con la restricción especificada y situarla cerca del ícono de la clase. Por ejemplo, se puede escribir `{temperatura = 35 o 39 o 42}` o bien `{capacidad = 20 o 30 o 40 kg}`.

UML permite definir restricciones para hacer las definiciones más explícitas. El método que emplea es un lenguaje completo denominado OCL (Object Constraint Language), lenguaje de restricción de objetos, que tiene sus propias reglas, términos y operadores (en el sitio oficial de OMG puede ver la documentación completa de la última versión de OCL).

Cristian Pilaparita 1095211-2202112346

16.2. Diseño y representación gráfica de clases en UML

En términos prácticos, una *clase* es un tipo definido por el usuario; son los bloques de construcción fundamentales de los programas orientados a objetos. Booch la denomina como "un conjunto de objetos que comparten una estructura y comportamiento comunes".

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto sabe que debe ejecutar. Estas acciones se conocen como *servicios, métodos* o *funciones miembro.* El término *función miembro* se utiliza, específicamente, en C++. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de ella; estos datos se conocen como *atributos* o *variables.* El término *atributo* se utiliza en análisis y diseño orientado a objetos y el término *variable* suele utilizarse en programas orientados a objetos.

El mundo real se compone de un gran número de objetos que interactúan entre sí; en muchas ocasiones, resultan muy complejos para poder ser entendidos en su totalidad. Por esta circunstancia suelen agruparse juntos elementos similares y con características comunes en función de las propiedades más sobresalientes e ignorando aquellas otras propiedades no tan relevantes. Éste es el proceso de abstracción ya citado anteriormente.

Este proceso suele comenzar con la identificación de características comunes de un conjunto de elementos y prosigue con la descripción concisa de estas características en lo que convencionalmente se conoce como **clase**.

Una clase describe el dominio de definición de un conjunto de objetos; cada uno pertenece a una clase. Las características generales están contenidas dentro de la clase y las características especializadas, en los objetos. Los objetos software se construyen a partir de las clases vía un proceso conocido como **instanciación**. De este modo un objeto es una **instancia** (ejemplar o caso) de una clase.

Así pues, una clase define la estructura y el comportamiento (datos y código) que serán compartidos por un conjunto de objetos. Cada objeto de una clase dada contiene la estructura (el estado) y el comportamiento definido por la clase y los objetos, como se ha definido anteriormente, suelen conocerse por instancias de una clase. Por consiguiente, una clase es una construcción lógica; un objeto tiene realidad física.

Una clase es una entidad que encapsula información y comportamiento.

Cuando se crea una clase, se especificará el código y los datos que la constituyen. De modo general, estos elementos se llaman *miembros* de la clase. De modo específico, los datos definidos en la clase se denominan *variables miembro* o *variables de instancia*. El código que opera sobre los datos se conoce como *métodos miembro* o simplemente *métodos*. En la mayoría de las clases, las variables de instancia son manipuladas o accedidas por los métodos definidos por esa clase. Por consiguiente, son los métodos los que determinan cómo se pueden utilizar los datos de la clase.

Las variables definidas en el interior de una clase se llaman variables de instancia debido a que cada instancia de la clase (es decir, cada objeto de la clase) contiene su propia copia de estas variables. Por consiguiente, los datos de un objeto son independientes y únicos de los datos de otro objeto.

Regla

- Los métodos y variables definidos en una clase se denominan *miembros* de la clase.
- En Java las operaciones se denominan *métodos*.
- En C++ las operaciones se denominan *funciones*.
- En *C#* las operaciones se denominan *métodos*, aunque también se admite el término *función*.

Cristian Pilapanta 1095211-2202112346

Dado que el propósito de una clase es encapsular complejidad, existen mecanismos para ocultar la complejidad de la implementación dentro de la clase. Cada método o variable de una clase se puede señalar como público o privado. La interfaz pública representa todo lo que los usuarios externos de la clase necesitan conocer o pueden conocer. Los métodos privados y los datos privados sólo pueden ser accedidos por el código que es miembro de la clase. Por consiguiente, cualquier otro código que no es miembro de la clase no puede acceder a un método privado o variable privada. Dado que los miembros privados de una clase sólo pueden ser accedidos por otras partes de su programa a través de los métodos públicos de la clase, se puede asegurar que no sucederá ninguna acción no deseada. Naturalmente, esto significa que la interfaz pública debe ser diseñada cuidadosamente para no exponer innecesariamente a la clase.

Una **clase** representa un conjunto de cosas o elementos que tienen un estado y un comportamiento común. Así, por ejemplo, `Volkswagen`, `Toyota`, `Honda` y `Mercedes` son todos coches que representan una clase denominada `Coche` (o `Carro`). Cada tipo específico de coche (carro) es una ***instancia*** de una clase o, dicho de otro modo, un **objeto**. Los objetos son miembros de clases y una clase es, por consiguiente, una descripción de un número de objetos similares. Así `Juanes`, `Carlos Vives`, `Shakira` y `Paulina Rubio` son miembros de la clase `CantantePop`, o de la clase `Músico`.

Un *objeto* es una *instancia* o ejemplar de una *clase*.

16.2.1. Representación gráfica de una clase

Una clase puede representar un concepto tangible y concreto, tal como un avión; puede ser abstracto, tal como un documento o un vehículo (como opuesto a una factura que es tangible), o puede ser un concepto intangible tal como *inversiones de alto riesgo*.

En UML 2.5.1 una clase se representa con una caja rectangular dividida en compartimentos, o secciones, o bandas. Un compartimento es el área del rectángulo donde se escribe información. El primer compartimento contiene el nombre de la clase, el segundo contiene los atributos y el tercero se utiliza para las operaciones. Se puede ocultar o quitar cualquier compartimento de la clase para aumentar la legibilidad del diagrama. Cuando no existe un compartimento no significa que esté vacío. Se pueden añadir compartimentos a una clase para mostrar información adicional, tal como excepciones o eventos, aunque no suele ser normal recurrir a incluir estas propiedades. UML propone que el nombre de una clase:

- *Comience con una letra mayúscula.*
- *Esté centrado en el compartimento (banda) superior.*
- *Sea escrito en un tipo de letra (fuente) negrita.*
- *Sea escrito en cursiva cuando la clase sea abstracta.*

Los atributos y operaciones son opcionales, aunque como se ha dicho anteriormente no significa que si no se muestran implique que estén vacíos. La figura 16.19 muestra unos diagramas, con la información oculta, muy fáciles de comprender.

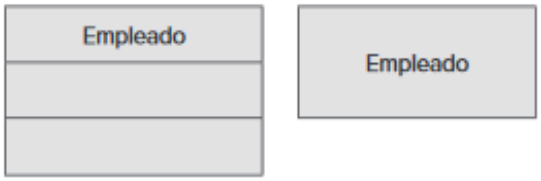


Figura 16.19. Clases en UML.

Cada clase se representa como un rectángulo subdivido en tres compartimentos o bandas. El primer compartimiento contiene el nombre de la clase; el segundo, los atributos; y el último, las operaciones. Por defecto, los atributos están ocultos y las operaciones son visibles. Estos compartimentos se pueden omitir para simplificar los diagramas.

Cristian Pilaparita 1095211-2202112347

EJEMPLOS

La clase ***Auto*** *contiene los atributos* `color`, `motor` *y* `velocidadMáxima`, *y puede agrupar las operaciones* `arrancar`, `acelerar` y `frenar`.

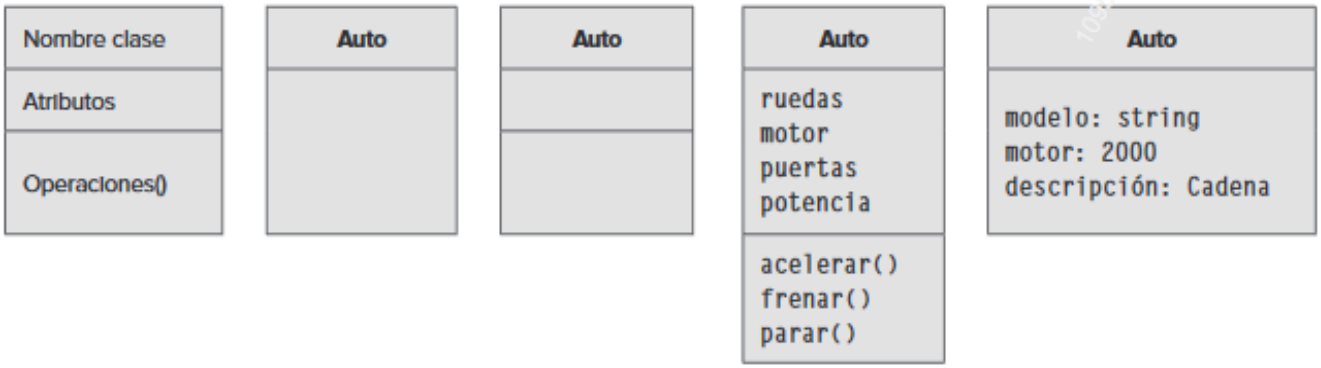
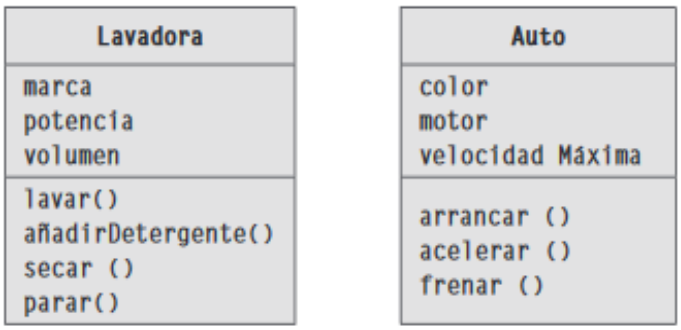


Figura 16.20. Cuatro formas diferentes de representar una clase.



a) *b*)

Figura 16.21. Diagrama de clases: *a*) `Lavadora`; *b*) `Auto`.

Reproductor/Grabador de video

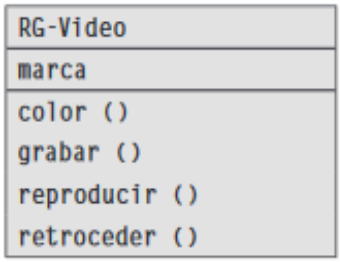


Figura 16.22. Clase RG de video.

Números complejos

Son números complejos aquellos que contienen una parte real y una parte imaginaria. Los elementos esenciales de un número complejo son sus coordenadas y se pueden realizar con ellos numerosas operaciones, tales como sumar, restar, dividir multiplicar, etcétera.

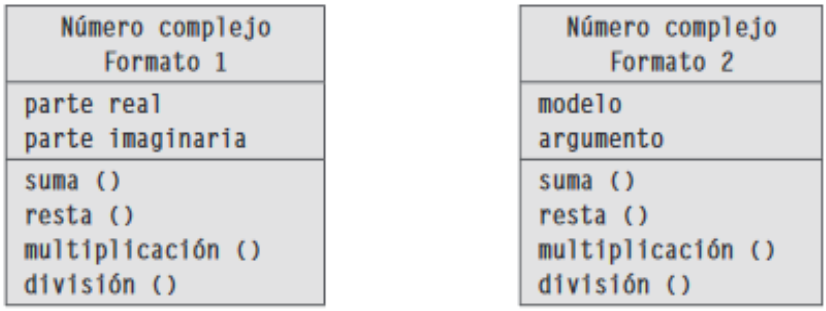


Figura 16.23. Clases número complejo.

Aparato de TV

Un aparato de TV es un dispositivo electrónico de complejidad considerable pero que puede utilizar adultos y niños. El aparato de TV ofrece un alto grado de abstracción merced a sus operaciones elementales.

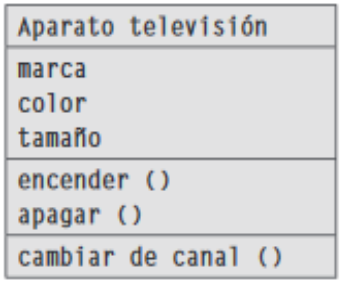


Figura 16.24. Clase Aparato de televisión.

Estructuras de datos

Representar los tipos abstractos de datos que manipulan las estructuras dinámicas de datos fundamentales: listas, pilas y colas.

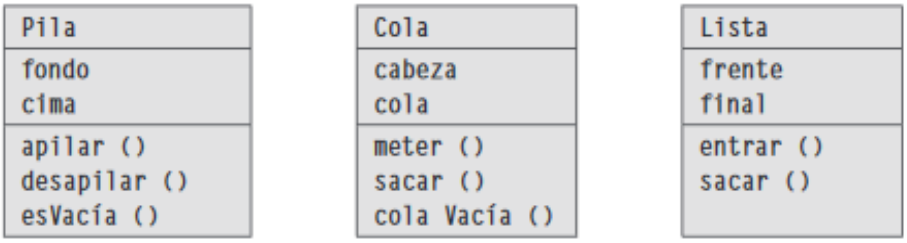


Figura 16.25. Clases de estructuras de datos.

Cristian Pilaparia 1095211-2202112347

16.2.2. Declaración de una clase

La declaración de una clase se divide en dos partes:

- La *especificación* de una clase describe el dominio de la definición y las propiedades de las instancias de esa clase, correspondiendo a la noción de un tipo como se define en los lenguajes de programación convencional.
- La *implementación* de una clase describe cómo se implementa la especificación y contiene los cuerpos de las operaciones y los datos necesarios para que las funciones actúen adecuadamente.

Los lenguajes modulares permiten la compilación independiente de la especificación y de la implementación de modo que es posible validar primero la consistencia de las especificaciones (también llamados interfaces) y a continuación validar la implementación en una etapa posterior. En lenguajes de programación, el concepto de tipo, descripción y módulo se integran en el concepto de clase con mayor o menor extensión.

- En C++, la clase se implementa directamente por una construcción sintáctica que incorpora el concepto de tipo, descripción y módulo. La clase se puede utilizar para obtener un módulo único añadiendo la palabra reservada `static` delante de todas las operaciones.
- En Java, la clase es la integración de los conceptos de tipo, descripción y módulo. También existe un concepto más general de módulos (el paquete) que puede contener varias clases.

La división entre especificación e implementación juega un papel importante en el nivel de abstracción y, en consecuencia, en el encapsulamiento. Las características más notables se describen en la especificación mientras que los detalles se circunscriben a la implementación.

Cristian Pilapanta 1095211-2202112347

Especificación de una clase

Antes de que un programa pueda crear objetos de cualquier clase, ésta debe ser *definida*; lo cual significa que se le debe dar un nombre, darle nombre a los elementos que almacenan sus datos y describir las funciones que realizarán las acciones consideradas en los objetos.

Las *definiciones* o *especificaciones* no son código de programa ejecutable. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer las funciones que éste utilizará. Normalmente se sitúan en archivos diferentes de los de código ejecutable, utilizando uno para cada clase. Se conocen como *archivos de cabecera* que se almacenan con un nombre de archivo con extensión *.h* en el caso del lenguaje de programación C++.

Formato

```
clase NombreClase
   lista_de_miembros
fin_clase
```

NombreClase	Nombre definido por el usuario que identifica a la clase (puede incluir letras, números y subrayados como cualquier identificador válido).
lista_de_miembros	Funciones y datos miembros de la clase obligatorio al final de la definición.

EJEMPLO 16.1

Definición en pseudocódigo de una clase llamada `Punto` *que contiene las coordenadas* `x` *y* `y` *de un punto en un plano.*

```
clase Punto
   //por omisión los atributos también son privados
   var
      privado entero: x, y     //coordenadas

   //por omisión los métodos también son públicos
   público entero función devolverX()
```

```
   //devuelve el valor de x
      inicio
         devolver(x)
      fin función

   público procedimiento fijarX(E entero: cx)
   //establece el valor de x
      inicio
         x ← cx
      fin procedimiento

   público entero función devolvery()
   //devuelve el valor de x
      inicio
         devolver(y)
      fin_función

   público procedimiento fijarY(E entero: cy)
   //establece el valor de x
      inicio
         y ← cy
      fin procedimiento
fin_clase
```

La definición de una clase no reserva espacio en memoria. El almacenamiento se asigna cuando se crea un objeto de una clase (*instancia* de una clase). Las palabras reservadas `público` y `privado` se llaman especificadores de acceso.

EJEMPLO 16.2

La definición en Java de la clase `Punto` *es:*

```
class Punto
{
   private int x, y;
   public int devolverX()
   {
      return (8x);
   }
   public void fijarX(int cx)
   {
      x = cx;
   }
   public int devolverY()
   {
      return (y);
   }
   public void fijarY(int cy)
   {
      y = cy;
   }
}
// programa que crea un objeto Punto
public class PruebaPunto
{
   public static void main(String[] args)
```

Cristian Pilapaña 1095211-2202112347

```
    {
        Punto p;
        p = new Punto();
        p.fijarX(4);
        p.fijarY(6);
    }
}
```

16.2.3. Reglas de visibilidad

Las reglas de visibilidad complementan o refinan el concepto de encapsulamiento. Los diferentes niveles de visibilidad dependen del lenguaje de programación con el que se trabaje, pero en general siguen el modelo de C++, aunque los lenguajes de programación Java y C# siguen también estas reglas. Estos niveles de visibilidad son:

- El nivel más fuerte se denomina nivel "privado"; la sección privada de una clase es totalmente opaca y sólo los amigos (término como se conoce en C++) pueden acceder a atributos localizados en la sección privada.
- Es posible aliviar el nivel de ocultamiento situando algunos atributos en la sección "protegida" de la clase. Estos atributos son visibles tanto para amigos como las clases derivadas de la clase servidor. Para las restantes clases permanecen invisibles.
- El nivel más débil se obtiene situando los atributos en la sección pública de la clase, con lo cual se hacen visibles a todas las clases.

Una clase se puede visualizar como en la figura 16.26.

Figura 16.26. Representación de atributos y operaciones con una clase.

El nivel de visibilidad se puede especificar en la representación gráfica de las clases en UML con los símbolos o caracteres #, + y – que corresponden con los niveles público, protegido y privado, respectivamente.

Reglas de visibilidad
+ Atributo público # Atributo protegido – Atributo privado
+ Operación pública ( ) # Operación protegida ( ) – Operación privada ( )

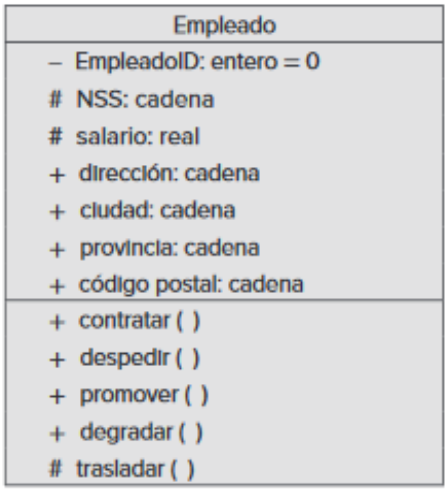
Los atributos privados están contenidos en el interior de la clase, ocultos a cualquier otra clase. Ya que los atributos están encapsulados dentro de una clase, se necesitará definir cuáles son las clases que tienen acceso a visualizar y cambiar los atributos. Esta característica se conoce como *visibilidad de los atributos*. Como ya se ha comentado, existen tres opciones de visibilidad (aunque algunos lenguajes como Java y C# admiten una cuarta opción de visibilidad denominada "paquete" o "implementación". El significado de cada visibilidad es el siguiente:

Cristian Pilaparia 1095211-2202112347

- ***Público***. El atributo es visible a todas las clases restantes; cualquier otra puede visualizar o modificar el valor del atributo. La notación UML de un atributo público es un signo más (+).
- ***Privado***. El atributo no es visible a ninguna otra clase. La notación UML de un atributo privado es un signo menos (–).
- ***Protegido***. La clase y cualquiera de sus descendientes tienen acceso a los atributos. La notación UML de un atributo protegido es el carácter "libra" o "almohadilla" (#).

EJEMPLO

Representar una clase Empleado:

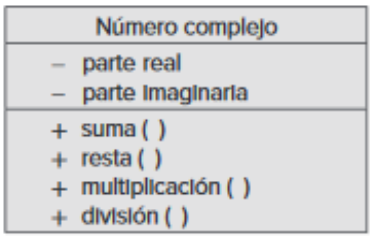


Reglas prácticas de visibilidad de atributos y operaciones

En general, se recomienda visibilidad privada o protegida para los atributos.

EJEMPLO

Representación de la clase Número complejo:



16.2.4. Sintaxis

Una clase se declara utilizando la palabra reservada **clase** del lenguaje UPSAM 2.0 (**class** en C++, Java y C#)

```
clase <nombre_de_clase>
   //Declaración de atributos
   const
      [privado | público | protegido]
         <tipo_de_dato> : <nombre_atributo> = <valor>
      ....
   var
      [estático] [público | privado | protegido]
```

Cristian Pilapanta 1095211-2202112347

```
        <tipo_de_dato> : <nombre_atributo> - [<valor_inicial>]
      ....
   //Declaraciones de métodos
   constructor <nombre_de_clase> ([<lista_de_parámetros_formales>])
      // Declaración de variables locales
      inicio
       ...
   fin_constructor
   ...
   [estático] [abstracto] [público | privado | protegido]
      <tipo_de_retorno> función <nombre_func>
      ([<lista_de_parámetros_formales>])
   inicio
       ...
       devolver(<resultado>)
   fin_función
   ...
   [estático] [abstracto] [público | privado | protegido]
      procedimiento <nombre_proc> ([<lista_de_parámetros_formales>])
    inicio
       ...
   fin_procedimiento
   destructor <nombre_de_clase> ()
      // Declaración de variables locales
      inicio
         ...
   fin_destructor
   ...
fin_clase
```

Cristian Pilapanta 1095211-2202112347

1095211220211234?

EJEMPLO

```
clase Mueble
   var
      público real: anchura
      público real: altura
      público real: profundidad
fin_clase
```

Nota

1. En Java la declaración de la clase y la implementación de los métodos se almacenan en el mismo sitio y no se definen por separado.
2. En C++, normalmente la declaración de la clase y la implementación de los métodos se definen por separado.

16.3. Declaración de objetos de clases

Una vez que una clase ha sido definida, un programa puede contener una *instancia*, denominada un *objeto de la clase*. Cuando se crea una clase se está creando un nuevo tipo de dato.

Formato

```
nombre_clase: identificador
```

EJEMPLO

```
Punto: p // Clase Punto, objeto p
```

En algunos lenguajes de programación se requiere un proceso en dos etapas para la declaración y asignación de un objeto.

1. Se declara una variable del tipo clase. Esta variable no define un objeto; es simplemente una variable que puede referir a un objeto.
2. Se debe adquirir una copia física, real del objeto y se asigna a esa variable utilizando el operador `nuevo` (en inglés, `new`); este operador asigna dinámicamente, es decir, en tiempo de ejecución, memoria para un objeto y devuelve una referencia al mismo. Esta referencia viene a ser una dirección de memoria del objeto asignado por `nuevo`. Esta referencia se almacena entonces en la variable.[2]

Sintaxis

```
varClase = nuevo nombreClase( )
```

EJEMPLO

Declarar una clase Libro y crear un objeto de esa clase.

```
clase Libro
   var
      real: anchura
      real: altura
      real: profundidad
   constructor Libro (real:a,b,c)
      inicio
         anchura ← a
         altura ← b
         profundidad ← c
   fin_constructor
fin_clase
...
```

Las dos etapas citadas anteriormente son:

```
Libro: milibro // declara una referencia al objeto
miLibro = nuevo Libro(5, 30, 20) // asigna un objeto Libro
```

Así, la definición de un objeto `Punto` es:

```
Punto: P
```

El *operador de acceso* a un miembro (.) selecciona un miembro individual de un objeto de la clase. Las siguientes sentencias, por ejemplo, crean un punto `P`, que fija su coordenada `x` y la visualiza.

```
Punto: p
P.fijarX (100);
Escribir "coordenada x es", P.devolverX()
```

[2] En Java, todos los objetos de una clase se deben asignar dinámicamente.

Cristian Pilapanta 1095211-2202112347

El operador punto se utiliza con los nombres de las funciones miembro para especificar que son miembros de un objeto.

Ejemplo: Clase DíaSemana, contiene una función Visualizar

```
DíaSemana: Hoy          // Hoy es un objeto
Hoy.Visualizar()        // ejecuta la función Visualizar
```

16.3.1. Acceso a miembros de la clase: encapsulamiento

Un principio fundamental en programación orientada a objetos es la *ocultación de la información*, que significa que determinados datos del interior de una clase no se pueden acceder por funciones externas a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o funciones privados sólo se puede acceder desde dentro de la clase. Por el contrario, los datos o funciones públicos son accesibles desde el exterior de la clase.

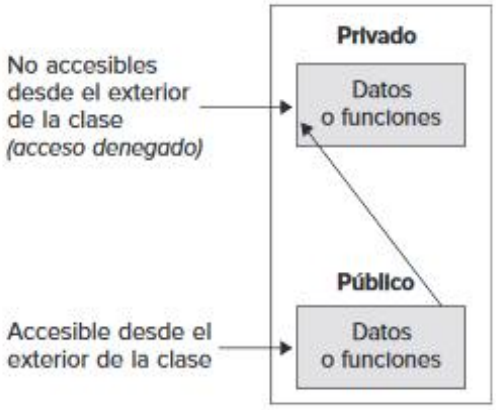


Figura 16.27. Secciones pública y privada de una clase.

Se utilizan tres diferentes *especificadores de acceso* para controlar el acceso a los miembros de la clase. Son público, privado y protegido. Se utiliza el formato general siguiente en definiciones de la clase.

Formato

```
const
   [privado | público | protegido] <tipo_dato>: nombre=<valor>
var
   [privado | público | protegido][estático] <tipo_dato>: <nombre>
                                  = <valor_inicial>
   [estático][abstracto][público | privado | protegido]
            <tipo_de_retorno> función <nombre> ([<parámetros>])
   [estático]ăbstracto][público | privado | protegido]
procedimiento <nombre> ([<parámetros>])
```

El especificador público define miembros públicos, que son aquellos a los que se puede acceder por cualquier función. A los miembros que siguen al especificador privado sólo se puede acceder por funciones miembro de la misma clase o por funciones y clases amigas.[3] A los miembros que siguen al especificador protegida se puede acceder

[3] Las funciones y clases amigas son propias de C++.

Cristian Pilapanta 1095211-2202112347

por funciones miembro de la misma clase o de clases derivadas de la misma, así como por amigas. Los miembros `público`, `protegido` y `privado` pueden aparecer en cualquier orden.

En la tabla 16.1 cada "x" indica que el acceso está permitido al tipo del miembro de la clase listado en la columna de la izquierda.

Tabla 16.1. Visibilidad.

Tipo de miembro	Miembro de la misma clase	Amiga	Miembro de una clase derivada	Función no miembro
`privado`	x	x		
`protegido`	x	x	x	
`público`	x	x	x	x

Si se omite el especificador, el acceso a los atributos se considera `privado` y a los métodos, `público`. En la siguiente clase `Estudiante`, por ejemplo, todos los datos son privados, mientras que las funciones miembro son públicas.

```
clase Estudiante
   var
      real: númId
      cadena: nombre
      entero: edad
   real función leerNúmId()
      inicio
         ...
      fin_función
      cadena función leerNombre()
      inicio
        ...
      fin_función
   entero función leerEdad()
      inicio
         ...
      fin_función
fin_clase
```

En C++ se establecen secciones públicas y privadas en las clases y el mismo especificador de acceso puede aparecer más de una vez en una definición de una clase, pero —en este caso— no es fácil de leer.

```
class Estudiante{
private:
   long númId;
public:
   long leerNúmId();
private:
   char nombre[40];
   int edad;
public:
   char* leerNombre();
   int LeerEdad();
};
```

El especificador de acceso se aplica a *todos* los miembros que vienen después de él en la definición de la clase (hasta que se encuentra otro especificador de acceso).

Cristian Pilaparita 1095211-2202112347

Aunque las secciones públicas y privadas pueden aparecer en cualquier orden, en C++ los programadores suelen seguir algunas reglas en el diseño que citamos a continuación y que usted puede elegir la que considere más eficiente.

1. Poner la sección privada primero, debido a que contiene los atributos (datos).
2. Se pone la sección pública primero, debido a que las funciones miembro y los constructores son la interfaz del usuario de la clase.

La regla 2 presenta realmente la ventaja de que los datos son algo secundario en el uso de la clase y con una clase definida adecuadamente en realidad no suele necesitarse ver cómo están declarados los atributos.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El principio de ocultación indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permite que los detalles de implementación de los objetos sean ignorados. Por consiguiente, las funciones miembro y los miembros datos de la *sección pública* forman la interfaz externa del objeto, mientras que los elementos de la *sección privada* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.

El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

Nota

Los lenguajes C++, Java y C# proporcionan un especificador de acceso, `protected`.

Cristian Pilapanta 1095211-2202112347

16.3.2. Declaración de métodos

Las clases normalmente constan de dos cosas: variables de instancia y métodos. Existen dos formatos para la declaración de los métodos, dependiendo de que se siga el modelo C++ *(función miembro)* o el modelo Java / C# *(método)*.

```
C++: tipo_retorno NombreClase:: nombreFunción(listaParametros)
     {
        // cuerpo de la función
     }
Java: tipo_retorno NombreClase(listaParametros)
     {
        // cuerpo del método
     }
```

Los métodos que tienen un tipo de retorno distinto de `void` devuelven un valor a la rutina llamadora utilizando el formato siguiente de la sentencia `return`:

```
return valor;
```

valor es el valor devuelto.

EJEMPLO

Clase Fecha en pseudocódigo:

```
//Pseudocódigo
clase Fecha
   var
      privado entero:  día, mes, anyo
```

```
   procedimiento fijarFecha(E entero: d, m, a)
      inicio
         día ← d
         mes ← m
         anyo ← a
   fin_procedimiento

   procedimiento mostrarFecha ()
      inicio
      escribir(día, '/', mes, '/', anyo)
   fin_procedimiento

fin_clase
```

Notación y asignación de valores a los miembros

En la sintaxis estándar para la referencia y asignación de valores a miembros de una clase se utilizan los siguientes formatos:

Formato:

```
nombre-objeto.nombre-atributo
nombre-objeto.nombre-función(parámetros)
```

Código Java

```
import java.io.*;
class Fecha
{
   protected int  día, mes, anyo;
   public void fijarFecha(int d, int m, int a)
   {
      día = d;
      mes = m;
      anyo = a;
   }
   public void mostrarFecha ()
   {
      System.out.println(día+"/"+mes+"/"+anyo);
   }
}

class PruebaFecha
{
   public static void main (String[] args)
   {
      Fecha f1, f2;
      f1 = new Fecha();
      f2 = new Fecha();

      f1.día = 15;
      f1.mes = 7;
      f1.anyo = 2002;
      f1. mostrarFecha();

      f2.fijarFecha(20, 8, 2002);
      f2.mostrarFecha();
   }
}
```

El resultado de la ejecución es:

```
15/7/2002
20/8/2002
```

A continuación se muestra un código en C++ donde se establecen los atributos como privados y las funciones miembro como públicas. A diferencia de Java, el modo de acceso predeterminado en C++ es `private`, otra diferencia es que en C++ las funciones miembro se declaran dentro de la clase pero, aunque puede hacerse dentro, suelen definirse fuera de la clase. En la definición de una función miembro fuera de la clase el nombre de la función ha de escribirse precedido por el nombre de la clase y por el operador binario de resolución de alcance (::).

```
include <iostream>

//Declaración de datos miembro y funciones miembro
class Fecha
{
      int día, mes, anyo;
   public:
      void fijarFecha(int, int, int);
      void mostrarFecha();
};

void Fecha::fijarFecha(int d, int m, int a)
{
   día = d;
   mes = m;
   anyo = a;
}
void Fecha::mostrarFecha ()
{
   cout << día <<"/"<< mes <<"/" << anyo;
}

//Prueba Fecha
int main()
{
   Fecha f;
   f.fijarFecha(20, 8, 2002);
   cout << "La fecha es "
   f.mostrarFecha();
   cout << endl;
   return 0;
}
```

El código en C# sería:

```
using System;
class Fecha
{
   int  día, mes, anyo;
   public void fijarFecha(int d, int m, int a)
   {
      día = d;
      mes = m;
      anyo = a;
   }
   public void mostrarFecha ()
   {
```

Cristian Pilaparita 1095211-2202112347
1095211220211234?

```
        Console.WriteLine(día+"/"+mes+"/"+anyo);
    }
}

class PruebaFecha
{
    public static void Main()
    {
        Fecha f;
        f = new Fecha();
        f.fijarFecha(20, 8, 2002);
        f.mostrarFecha();
    }
}
```

Los campos de datos en este último ejemplo también son privados, por lo que, con el código especificado, la sentencia

```
Console.WriteLine(f.día);
```

daría error.

Cristian Pilaparita 1095211-2202112347

16.3.3. Tipos de métodos

Los métodos que pueden aparecer en la definición de una clase se clasifican en función del tipo de operación que representan. Estos métodos tienen una correspondencia con los tipos de mensajes que se pueden enviar entre los objetos de una aplicación, como por otra parte era lógico pensar.

- *Constructores* y *destructores*, son funciones miembro a las que se llama automáticamente cuando un operador se crea o se destruye.
- *Selectores*, que devuelven los valores de los miembros dato.
- *Modificadores* o *mutadores*, que permiten a un programa cliente cambiar los contenidos de los miembros dato.
- *Operadores*, que permiten definir operadores estándar para los objetos de las clases.
- *Iteradores*, que procesan colecciones de objetos, tales como arrays y listas.

16.4. Constructores

Un **constructor** es un método que tiene el mismo nombre que la clase y cuyo propósito es inicializar los miembros datos de un nuevo objeto que se ejecuta automáticamente cuando se crea un objeto de una clase. Sintácticamente es similar a un método. Dependiendo del número y tipos de los argumentos proporcionados, una función o método constructor se llama automáticamente cada vez que se crea un objeto. Si no se ha escrito ninguna función constructor en la clase, el compilador proporciona un constructor por defecto. A su rol como inicializador, un constructor puede también añadir otras tareas cuando es llamado.

Un constructor tiene el mismo nombre que la propia clase. Cuando se define un constructor no se puede especificar un valor de retorno, ni incluso `nada (void)`; un constructor nunca devuelve un valor; sin embargo, tomar cualquier número de parámetros (cero o más).

Reglas

1. El constructor tiene el mismo nombre que la clase.
2. Puede tener cero o más parámetros.
3. No devuelve ningún valor.

EJEMPLO 16.3

La clase `Rectángulo` *tiene un constructor con cuatro parámetros. El código se muestra en C++.*

```
class Rectángulo
{
   private:
      int izdo;
      int superior;
      int dcha;
      int inferior;
public:
   //Constructor
   Rectángulo(int i, int s, int d, int inf);
   //Definiciones de otras funciones miembro
};
```

Cuando se define un objeto, se pasan los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada normal de la función:

```
Rectángulo Rect(25, 25, 75, 75);
```

Esta definición crea una instancia del objeto `Rectángulo` e invoca al constructor de la clase pasándole los parámetros con valores especificados.

Se puede también pasar los valores de los parámetros al constructor cuando se crea la instancia de una clase utilizando el operador `new`:

```
Rectángulo *Crect = new Rectángulo(25, 25, 75, 75);
```

El operador `new` invoca automáticamente al constructor del objeto que se crea (ésta es una ventaja importante de utilizar `new` en lugar de otros métodos de asignación de memoria tales como la función `malloc`).

Cristian Pilaparita 1095211-2202112347

16.4.1. Constructor por defecto (predeterminado)

Un constructor que no tiene parámetros se llama *constructor por defecto;* y normalmente inicializa los miembros dato asignándoles valores por defecto (valores predeterminados o preasignados).

EJEMPLO 16.4

El constructor por defecto inicializa x y y a 0.

```
// Clase Punto implementada en C++
class Punto
{
   public:
      Punto
      {
         x = 0;
         y = 0;
      }

   private
      int x;
      int y;
}
```

Una vez que se ha declarado un constructor, cuando se declara un objeto `Punto` sus miembros dato se inicializan a 0. Ésta es una buena práctica de programación.

```
Punto P1   // P1.x = 0, P1.y = 0
```

Si `Punto` se declara dentro de una función, su constructor se llama tan pronto como la ejecución del programa alcanza la declaración de `Punto`:

```
void FuncDemoConstructorD()
{
   Punto pt; // llamada al constructor
   // ...
}
```

Regla

C++ crea automáticamente un constructor por defecto cuando no existen otros constructores. Sin embargo, tal constructor no inicializa los miembros dato de la clase a un valor previsible, de modo que siempre es conveniente al crear su propio constructor por defecto darle la opción de inicializar los miembros dato con valores previsibles.

Precaución

Tenga cuidado con la escritura de la siguiente sentencia:

```
Punto P();
```

Aunque parece que se realiza una llamada al constructor por defecto, lo que se hace es declarar una función de nombre `P` que no tiene parámetros y devuelve un resultado de tipo `Punto`.

Cristian Pilaparita 1095211-2202112347

Formato

1. Un constructor debe tener el mismo nombre que la clase a la cual pertenece.
2. No tiene ningún tipo de retorno (ni incluso `void`).

```
// Programa en Java
import java.io.*;
class Rectángulo
{
   private double longitud, anchura;
   // constructor
   Rectángulo(double l, double a)
   {
      longitud = l;
      anchura = a;
   }
   double perímetro()
   {
      return 2*(longitud+anchura);
   }
}
class PruebaRectángulo
{
   public static void main(String[] args)
   {
```

```
        Rectángulo  r;
        r = new Rectángulo(3.5, 6.5);
        System.out.println(r.perímetro()) ;
    }
}
```

```
//Código en Turbo Pascal 7.0
unit uobjeto2;
interface
   type
      Rectángulo = object
         constructor Rectángulo (l, a: real);
         function perímetro: real;
      private
         longitud, anchura: real;
   end;
implementation
   {pueden omitirse los parámetros, pues ya están
      especificados en la declaración}
   constructor Rectángulo.Rectángulo;
      begin
      longitud := l;
      anchura := a
   end;
  function Rectángulo.perímetro;
   begin
      perímetro := 2*(longitud + anchura)
   end;
end.
```

```
program PruebaR;
 uses uobjeto2;
 var
   p: ^Rectángulo;
 begin
   {new crea un objeto en el montículo y lo inicializa al
    llevar como segundo parámetro el constructor}
   new (p,rectángulo(3.5, 6.5));
   writeln('El perímetro es ', p^.perímetro:0:2);
 end.
```

También es válido crear una instancia del objeto `Rectángulo` sin emplear `new` e invocar al constructor utilizando una sintaxis similar a la empleada para llamar a un procedimiento.

```
program PruebaR;
 uses uobjeto2;
 var
   r: Rectángulo;
 begin
   r.Rectángulo(3.5, 6.5);
```

Cristian Pilapanta 1095211-2202112347
1095211220211234?

```
    writeln('El perímetro es ', r.perímetro:0:2);
end.
```

La clase `Rectángulo` en C++:

```
class Rectángulo
{
   int longitud;
   int anchura;
  public:
   Rectángulo(int l, int a);
   //definiciones de otras funciones miembro
}
```

Cuando en una clase no se declara ningún constructor, el compilador crea uno por defecto. El constructor por defecto inicializa todas las variables instancia a cero o por el contrario también se refiere a aquel constructor que no requiere la declaración de ningún parámetro o porque a todos los parámetros se les ha dado un valor por defecto.

Nota

Un *constructor* es cualquier función que tiene el mismo nombre que su clase. El propósito principal de un constructor es inicializar las variables miembro de un objeto cuando éste se crea. Por consiguiente, un constructor se llama automáticamente cuando se declara un objeto.

En general, una clase puede contener múltiples constructores pero se diferencian entre sí en la lista de parámetros. Cada constructor se debe declarar sin ningún tipo de dato de retorno (ni incluso void).

Cristian Pilapanta 1095211-2202112347

16.5. Destructores

La contrapartida a un constructor es un destructor Los destructores son funciones (métodos) que tienen el mismo nombre de la clase al igual que los constructores, pero para distinguirlos sintácticamente se les precede por una tilde (–) o por la palabra reservada `destructor`.

EJEMPLO

```
~Fecha ()
```

Al igual que sucede con los constructores, se proporciona un constructor por defecto en el caso de que no se incluya explícitamente en la declaración de la clase. Al contrario que los constructores, sólo puede haber un destructor por clase; esto se debe a que los destructores no pueden tener argumentos ni devolver valores.

Los destructores se llaman automáticamente siempre que un objeto deje de existir y su objetivo es limpiar cualquier efecto no deseado que haya podido dejar el objeto.

Regla

Una función destructor se llama a la vez que un objeto sale fuera de ámbito (desaparece).

Los destructores deben tener el mismo nombre que su clase pero suelen ir precedidos de una tilde.

Sólo puede haber un destructor por clase.

Un destructor no tiene argumentos ni devuelve ningún valor. Si no se incluye ningún destructor en la clase, el compilador proporciona un destructor por defecto.

EJEMPLO 16.5

Se declara una clase con constructor y destructor.

```
//C++
class Demo
{
   int datos;
public:
   Demo() {datos = 0;}          // constructor
   ~Demo() {}                   // destructor
};
```

Regla

- Los destructores no tienen valor de retorno.
- Tampoco tienen argumentos.

El uso más frecuente de un destructor es liberar memoria que fue asignada por el constructor. Si un destructor no se declara explícitamente, se crea uno vacío automáticamente. Si un objeto tiene ámbito local, su destructor se llama cuando el control pasa fuera de su bloque de definición.

Regla en C++

Si un objeto tiene ámbito de archivo, el destructor se llama cuando termina el programa principal (`main`). Si un objeto se asignó dinámicamente (utilizando `new` y `delete`), el destructor se llama cuando se invoca el operador `delete`.

En C# la memoria se libera automáticamente, a través de un recolector de basura (*Garbage Collector*), que llama a los destructores a partir del momento en el que se sabe que un objeto ya no va a ser utilizado. El recolector de basura invoca al destructor, que es el que sabe cómo liberar el recurso, en el momento que considera oportuno. En el bloque de un destructor deben especificarse las instrucciones especiales que deben ser ejecutadas al destruir un objeto de la clase.

EJEMPLO 16.6

Programa en el que se activa el constructor y destructor de una clase C#.

```
//C#
using System;
class Punto
{
   int x, y;
   public Punto(int cx, int cy) {
      x = cx;
      y = cy;
   }
```

Cristian Pilapanta 1095211-2202112348

```
    ~Punto() {
       Console.WriteLine("Se ha llamado al destructor de Punto");
    }
}
class PruebaDestructores
{
    public static void main()
    {
       Punto p = new Punto(3,4);
       p = null;
       //la siguiente instrucción fuerza la recolección de basura
       GC.Collect();
       //Hace que el hilo actual espere a que la cola de
       //destructores quede vacía
       GC.WaitForPendingFinalizers();
    }
}
```

Java también tiene recolección automática de basura, siendo el método `finalize` el que se redefine para efectuar operaciones especiales de limpieza.

16.6. Implementación de clases en C++

El código fuente para la implementación de funciones miembro de una clase es código ejecutable. Se almacena, por consiguiente, en archivos de texto con extensiones `.cp` o `.cpp`. Normalmente se sitúa la implementación de cada clase en un archivo independiente.

Cada implementación de una función tiene la misma estructura general. Obsérvese que una función comienza con una línea de cabecera que contiene, entre otras cosas, el nombre de la función y su cuerpo está acotado entre una pareja de signos llave. Las clases pueden proceder de diferentes fuentes:

- Se pueden declarar e implementar sus propias clases. El código fuente siempre estará disponible.
- Se pueden utilizar clases que hayan sido escritas por otras personas o incluso que se han comprado. En este caso se puede disponer del código fuente o estar limitado a utilizar el código objeto de la implementación.
- Se puede utilizar clases de las bibliotecas del programa que acompañan a su software de desarrollo C++. La implementación de estas clases se proporciona normalmente como código objeto.

En cualquier forma, se debe disponer de las versiones de texto de las declaraciones de clase para que pueda utilizarlas su compilador.

16.6.1. Archivos de cabecera y de clases

Las declaraciones de clases se almacenan normalmente en sus propios archivos de código fuente, independientes de la implementación de sus funciones miembro. Éstos son los *archivos de cabecera* que se almacenan con una extensión *.h* en el nombre del archivo.

El uso de archivos de cabeceras tiene un beneficio muy importante: "Se puede tener disponible la misma declaración de clases a muchos programas sin necesidad de duplicar la declaración". Esta propiedad facilita la reutilización en programas C++.

Para tener acceso a los contenidos de un archivo de cabecera, un archivo que contiene la implementación de las funciones de la clase declaradas en el archivo de cabecera o un archivo que crea objetos de la clase declarada en el archivo de cabecera *incluye* (`include`), o mezcla, el archivo de cabecera utilizando una *directiva de compilador*, que es una instrucción al compilador que se procesa durante la compilación. Las directivas del compilador comienzan con el signo "almohadilla" (#).

Cristian Pilaparta 1095211-2202112348

```
// Declaración de una clase almacenada en Demo1.h
clase Demo1
public:
   Demo1();
   void Ejecutar();
fin_clase
```

```
// Declaración de la clase edad almacenada en edad.h
class edad
{
   private:
      int edadHijo, edadPadre, edadMadre;
  public:
     edad();
      void iniciar(int, int, int);
      int obtenerHijo();
      int obtenerPadre();
      int obtenerMadre();
};
```

Figura 16.28. Listado de declaraciones de clases.

Cristian Pilapaña 1095211-2202112348

La directiva que mezcla el contenido de un archivo de cabecera en un archivo que contiene el código fuente de una función es:

```
#include nombre-archivo
```

Opciones de compilación

La mayoría de los compiladores soporta dos versiones ligeramente diferentes de esta directiva. La primera instruye al compilador a que busque el archivo de cabecera en un directorio de disco que ha sido designado como el depósito de archivos de cabecera.

EJEMPLO

```
#include <iostream>
```

utiliza la biblioteca de clases que soporta E/S.

La segunda versión se produce cuando el archivo de cabecera está en un directorio diferente; entonces, se pone el nombre del camino entre dobles comillas.

EJEMPLO

```
#include "/mi.cabecera/cliente.h"
```

16.6.2. Clases compuestas

Una *clase compuesta* es aquella que contiene miembros dato que son, asimismo, objetos de clases. Antes de que el cuerpo de un constructor de una clase compuesta, se deben construir los miembros dato individuales en su orden de declaración.

La clase Estudiante contiene miembros dato de tipo Expediente y Dirección:

```
// código en C#
class Expediente
{
   //...
}
class Dirección
{
   //...
}
class Estudiante
{
   string id;
   Expediente exp;
   Dirección dir;
   float notaMedia;

   public Estudiante()
   {
      PonerId ("");
      PonerNotaMedia(0.0F);
      dir = new Dirección();
      exp = new Expediente();
   }
   public void PonerId (string v)
   {
      id = v;
   }
   public void PonerNotaMedia(float v)
   {
      notaMedia = v;
   }
   public void Mostrar()
   {
   }
}
```

Aunque Estudiante contiene Expediente y Dirección, el constructor de Estudiante no tiene acceso a los miembros privados o protegidos de Expediente o Dirección. Cuando un objeto Estudiante sale fuera de alcance, se llama a su destructor. Aunque generalmente el orden de las llamadas a destructores a clases compuestas es exactamente el opuesto al orden de llamadas de constructores, en C++ no se tiene control sobre cuándo un destructor va a ser ejecutado, ya que son llamados automáticamente por el recolector de basura.

16.7. Recolección de basura

Como los objetos se asignan dinámicamente, cuando éstos se destruyen será necesario verificar que la memoria ocupada por ellos ha quedado liberada para usos posteriores. El procedimiento de liberación es distinto según el tipo de lenguaje utilizado.

En C++ los objetos asignados dinámicamente se deben liberar utilizando un operador delete. Por el contrario, Java y C# tienen un enfoque diferente. Manejan la liberación de memoria de modo automático. La técnica que utilizan se denomina recolección de basura (*garbage collection*). Su funcionamiento es el siguiente: cuando no existe ninguna referencia a un objeto, se supone que ese objeto ya no se necesita, y la memoria ocupada por él puede ser recuperada (liberada). No hay necesidad de destruir objetos explícitamente como hace C++. La recolección de ba-

Cristian Pilapanta 1095211-2202112348
1095211220211234

sura sólo ocurre esporádicamente durante la ejecución de su programa. No sucede simplemente porque los objetos dejen de ser utilizados.

16.7.1. El método `finalize ()`

En ocasiones se necesita que un objeto realice alguna acción cuando se destruye. Por ejemplo, un objeto contiene algún recurso no-Java tal como un manejador de archivos o una fuente de caracteres de Windows, entonces puede desear asegurarse que esos recursos se liberan antes de que se destruya el objeto. El mecanismo que utilizan algunos lenguajes, como es el caso de C# y Java, se llama *finalización*. Utilizando este mecanismo se pueden definir acciones específicas que ocurrirán cuando un objeto está a punto de ser liberado por el recolector de basura.

Para añadir un *finalizador* a una clase, basta con definir el método `finalize()` (en Java). Dentro del método `finalize()` se especificarán aquellas acciones que se deben ejecutar antes de que se destruya un objeto. El recolector de basura se ejecuta periódicamente comprobando qué objetos no están siendo utilizados por ningún estado de ejecución o indirectamente referenciados por otros objetos.

Formato

```
protegido destructor <nobreclase>()
inicio
fin_destructor
```

La palabra reservada protegido (`protected`) es un especificador que previene el acceso al destructor por código definido al exterior de su clase. Si no se especifica, nada es **protegido**.

Importante

En Java `finalize()` sólo se llama antes de la recolección de basura. Si no se llama cuando un objeto sale fuera de ámbito, significa que no se puede conocer cuándo —o incluso si— se ejecutará `finalize()`. En consecuencia, es importante que su programa proporcione otros medios de liberar recursos del sistema.

Nota C++/Java

C++ permite definir un destructor para una clase que se llama cuando un objeto sale fuera de ámbito (se destruye). Java no soporta destructores. La idea aproximada del destructor en Java es el método `finalize()` y sus tareas son realizadas por el subsistema de recolección de basura.

CONCEPTOS CLAVE

- Clase abstracta
- Clase compuesta
- Comunicación entre objetos
- Constructor
- Declaración de acceso
- Destructor
- Encapsulamiento
- Función miembro
- Función virtual
- Herencia
- Instancia
- Mensaje
- Método
- Miembro dato
- Objeto
- Ocultación de la información
- Privada
- Protegida
- Pública
- Relación **es-un**
- Relación **tiene-un**

Cristian Pilapanta 1095211-2202112348

RESUMEN

Una clase es un conjunto de objetos que constituyen instancias de la clase, cada una de las cuales tienen la misma estructura y comportamiento. Una clase tiene un nombre, una colección de operaciones para manipular sus instancias y una representación. Las operaciones que manipulan las instancias de una clase se llaman *métodos*. El estado o representación de una instancia se almacena en variables de instancia. Estos métodos se invocan mediante el envío de *mensajes* a instancias. El envío de mensajes a objetos (instancias) es similar a la llamada a procedimientos en lenguajes de programación tradicionales.

Los principales puntos clave tratados son:

- La programación orientada a objetos incorpora estos seis componentes importantes:
  - Objetos.
  - Clases.
  - Métodos.
  - Mensajes.
  - Herencia.
  - Polimorfismo.
- Un objeto se compone de datos y funciones que operan sobre esos objetos.
- La técnica de situar datos dentro de objetos de modo que no se puede acceder directamente a los datos se llama *ocultación de la información.*
- Los programas orientados a objetos pueden incluir *objetos compuestos*, que son objetos que contienen otros, anidados o integrados en ellos mismos.
- Una clase es una descripción de un conjunto de objetos. Una instancia es una variable de tipo objeto y un objeto es una instancia de una clase.
- La herencia es la propiedad que permite a un objeto pasar sus propiedades a otro objeto, o dicho de otro modo, un objeto puede heredar de otro objeto.
- Los objetos se comunican entre sí pasando mensajes.
- La clase padre o ascendiente se denomina *clase base* y las clases descendientes, clases derivadas.
- La reutilización de software es una de las propiedades más importantes que presenta la programación orientada a objetos.
- El polimorfismo es la propiedad por la cual un mismo mensaje puede actuar de diferente modo cuando actúa sobre objetos diferentes ligados por la propiedad de la herencia.
- Una **clase** es un tipo de dato definido por el usuario que sirve para representar objetos del mundo real.
- Un objeto de una clase tiene dos componentes —un conjunto de atributos y un conjunto de comportamientos (operaciones)—. Los atributos se llaman miembros dato y los comportamientos se llaman funciones miembro.

```
clase círculo
   var
      público real: x_centro,
      y_centro, radio
   público real función superficie()
      inicio
      ...
      fin función
   fin_clase
```

- Cuando se crea un nuevo tipo de clase, se deben realizar dos etapas fundamentales: determinar los atributos y el comportamiento de los objetos.
- Un objeto es una instancia de una clase.

```
círculo un_círculo
```

- Una declaración de una clase se divide en tres secciones: pública, privada y protegida. La sección pública contiene declaraciones de los atributos y el comportamiento del objeto que son accesibles a los usuarios del objeto. La declaración de los constructores se recomienda en la sección pública. La sección privada contiene las funciones miembro y los miembros dato que son ocultos o inaccesibles a los usuarios del objeto. Estas funciones miembro y atributos dato son accesibles sólo por la función miembro del objeto.
- El acceso a los miembros de una clase se puede declarar como *privado* (`private`, por defecto), *público* (`public`) o *protegido* (`protected`).

```
clase Círculo
   var
      privado real: centro_x,
      centro_y, radio
   público real función Superficie ()
      inicio
          ...
          devolver (...)
      fin_función
   público procedimiento Fijar-Centro
```

Cristian Pilapaña 1095211-2202112348

```
(E real: x, y)
   inicio
   ...
   fin_procedimiento
público procedimiento Fijar-Radio
(E real: r)
   inicio
      ...
   fin_procedimiento
público real función DevolverRadio ()
   inicio
      ...
   devolver (...)
   fin_función
fin_clase
```

Los miembros dato `centro_x`, `centro_y` y `radio` son ejemplos de ocultación de datos.

- El procedimiento fundamental de especificar un objeto es

```
círculo :c        // un objeto
```

y para especificar un miembro de una clase

```
radio = 10.0      // Miembro de la clase
```

El operador de acceso a miembro (el operador punto).

```
c.radio = 10.0;
```

- Un **constructor** es una función miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobrecargado.

```
clase Complejo
   ...
   constructor Complejo (real:x,y)
   inicio
   ...
   fin_constructor
```

- Un **constructor** es una función miembro especial que se invoca cuando se crea un objeto. Se utiliza normalmente para inicializar los atributos de un objeto. Los argumentos por defecto hacen al constructor más flexible y útil.
- El proceso de crear un objeto se llama *instanciación* (creación de instancia).
- Un **destructor** es una función miembro especial que se llama automáticamente siempre que se destruye un objeto de la clase.

```
destructor Complejo ()
inicio
   ...
fin_destructor
```

Cristian Pilapanta 1095211-2202112348

EJERCICIOS

16.1. Consideremos una pila como un tipo abstracto de datos. Se trata de definir una clase que implementa una pila de 100 caracteres mediante un array. Las funciones miembro de la clase deben ser:

`meter, sacar, pilavacía y pilallena`

16.2. Reescribir la clase pila que utilice una lista enlazada en lugar de un array (sugerencia: utilice otra clase para representar los modos de la lista).

16.3. Crear una clase llamada `hora` que tenga miembros datos separados de tipo int para horas, minutos y segundos. Un constructor inicializará este dato a 0 y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo hora pasados como argumentos. Una función principal `main()` crea dos objetos inicializados y uno que no está inicializado. Sumar los dos valores inicializados y dejar el resultado en el objeto no inicializado. Por último, visualizar el valor resultante.

16.4. Crear una clase llamada `empleado` que contenga como miembro dato el nombre y el número de empleado y como funciones miembro `leerdatos()` y `verdatos()` que lean los datos del teclado y los visualice en pantalla, respectivamente.

Escribir un programa que utilice la clase, creando un array de tipo empleado y luego llenándolo con datos correspondientes a 50 empleados. Una vez rellenado el array, visualizar los datos de todos los empleados.

16.5. Se desea realizar una clase Vector3d que permita manipular vectores de tres componentes (coordenadas `x`, `y`, `z`) de acuerdo con las siguientes normas:

- Sólo posee una función constructor y es en línea.
- Tiene una función miembro `igual` que permite saber si dos vectores tienen sus componentes o coordenadas iguales (la declaración de `igual` se realizará utilizando: *a*) transmisión por valor; *b*) transmisión por dirección; *c*) transmisión por referencia).

16.6. Incluir en la clase `Vector3d` del ejercicio anterior una función miembro denominada `normamáx` que permita obtener la norma mayor de dos vectores. (Nota: La norma de un vector $v = x, y, z$ es $x^2 + y^2 + z^2$ o bien `x*x + y*y + z*z`.)

16.7. Incluir en la clase `Vector3d` del ejercicio anterior las funciones miembros `suma` (suma de dos vectores), `productoescalar` (producto escalar de dos vectores: `v1= x1, y1, z1; v2 = x2, y2, z2; v1* v2 = x1* x2 + y1 * y2 + z1 * z2`).

16.8. Realizar una clase `Complejo` que permita la gestión de números reales (un número complejo = dos números reales `real (doble)`: una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- Una función `establecer()` permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.
- Una función `imprimir()` realiza la visualización formateada de un `Complejo`.
- Dos funciones `agregar()` (sobrecargadas) permiten añadir, respectivamente, un `Complejo` a otro y añadir dos componentes `real` a un `Complejo`.

16.9. Escribir una clase `Conjunto` que gestione un conjunto de enteros `(entero)` con ayuda de una tabla de tamaño fino (un *conjunto* contiene una lista no ordenada de elementos y se caracteriza por el hecho de que cada elemento es único: no se debe encontrar dos veces el mismo valor en la tabla). Las operaciones a implementar son las siguientes:

- La función `vacía()` vacía el conjunto.
- La función `agregar()` añade un entero al conjunto.
- La función `eliminar()` retira un entero del conjunto.
- La función `copiar()` recopila un conjunto en otro.
- La función `es_miembro()` reenvía un valor booleano (lógicos que indica si el conjunto contiene un elemento, un entero dado).
- La función `es_igual()` reenvía un valor *booleano* que indica si un conjunto es igual a otro.
- La función `imprimir()` realiza la visualización formateada del conjunto.

16.10. Crear una clase `Lista` que realice las siguientes tareas:

- Una lista simple que contenga cero o más elementos de algún tipo específico.
- Crear una lista vacía.
- Añadir elementos a la lista.
- Determinar si la lista está vacía.
- Determinar si la lista está llena.
- Acceder a cada elemento de la lista y realizar alguna acción sobre ella.

16.11. Añadir a la clase `Hora` del ejercicio 16.3 las funciones de acceso, una función `adelantar(int h, int m, int s)` para adelantar la hora actual de un objeto existente, una función `reiniciar(int h, int m, int s)` que reinicia la hora actual de un objeto existente y una función `imprimir()`.

16.12. Añadir a la clase `Complejo` del ejercicio 16.8 las operaciones:

- Suma: `a + c = (A+, (B + D)i)`.
- Resta: `a − c = (A − C, B − D)i)`.
- Multiplicación: `a * c = (A * C − B * D, (A * D + B * C)i)`.
- Multiplicación: `x * c = (x * C, x * Di)`, donde *x* es `real`.
- Conjugado: `~a = (A;−Bi)`.

16.13. Implementar una clase `Random` (aleatoria) para generar números pseudoaleatorios.

16.14. Implementar una clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, funciones de acceso, una función `reiniciar (int d, int m, int a)` para reiniciar la fecha de un objeto existente, una función `adelantar(int d, int m, int a)` para avanzar a una fecha existente (día, `d`, mes, `m`, y año, `a`) y una función `imprimir()`. Utilizar una función de utilidad `normalizar()` que asegure que los miembros dato están en el rango correcto `1 ≤ año, 1 ≤ mes ≤ 12, día ≤ días (Mes)`, donde `días(Mes)` es otra función que devuelve el número de días de cada mes.

Cristian Pilapanta 1095211-2202112348

16.15. Ampliar el programa anterior de modo que pueda aceptar años bisiestos. **Nota**: un año es bisiesto si es divisible por 400, o si es divisible por 4 pero no por 100. Por ejemplo, el año 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.

LECTURAS RECOMENDADAS

Booch, G.: *Object-Oriented Analysis and Design with Applications*, Reedwood City, CA (USA): BenjaminCummings, 1994.

Joyanes, Luis: *Programación orientada a objetos*, 2a. edición, Madrid, McGraw-Hill, 1998.

Joyanes, Luis. *Programación en C, C++, Java y UML*. 2a. edición, 2014. México, McGrawHill, 2014.

Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F., y Lorensen, W.: *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ (USA), Prentice-Hall, 1991.

Wirfs-Brock, R.; Wilkerson, B., y Wiener, L.: *Designing Object-Oriented Software*, Englewood Cliffs, NJ (USA), Prentice-Hall, 1990.

Cristian Pilapanta 1095211-2202112348

CAPÍTULO 17

Relaciones entre clases: delegaciones, asociaciones, agregaciones, herencia

17.1. Relaciones entre clases
17.2. Dependencia
17.3. Asociación
17.4. Agregación
17.5. Jerarquía de clases: generalización y especialización
17.6. Herencia: clases derivadas
17.7. Accesibilidad y visibilidad en herencia
17.8. Un caso de estudio especial: herencia múltiple
17.9. Clases abstractas
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

Cristian Pilapanta 1095211-2202112348

1095211220211234B

INTRODUCCIÓN

En este capítulo se introducen los conceptos fundamentales de relaciones entre clases. Las relaciones más importantes soportadas por la mayoría de las metodologías de orientación a objetos y en particular por UML son: asociación, agregación y generalización/especialización. En el capítulo se describen estas relaciones, así como las notaciones gráficas correspondientes en UML.

De modo especial se introduce el concepto de *herencia* como exponente directo de la relación de generalización/especialización y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes *orientados a objetos* de los lenguajes *basados en objetos* y *lenguajes estructurados*. La *herencia* es la propiedad que permite definir nuevas clases usando como base a clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, de bajo coste, adaptable y reutilizable.

17.1. Relaciones entre clases

Una relación es una conexión semántica entre clases. Permite que una clase conozca sobre los atributos, operaciones y relaciones de otras. Las clases no actúan aisladas entre sí, al contrario, están relacionadas unas con otras. Una puede ser un tipo de otra clase —generalización— o bien puede contener objetos de otra clase de varias formas posibles, dependiendo de la fortaleza de la relación entre las dos.

La fortaleza de una relación de clases (Miles, Hamilton 2006) se basa en el modo de dependencia de las clases implicadas en las relaciones entre ellas. Dos clases que son fuertemente dependientes una de otra se dice que están *acopladas fuertemente* y, en caso contrario, están *acopladas débilmente*.

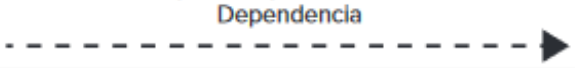
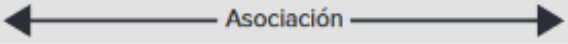
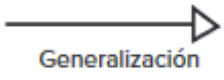
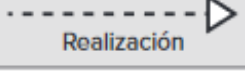
Figura 17.1. Relaciones entre clases.

Las relaciones entre clases se corresponden con las relaciones entre objetos físicos del mundo real, o bien objetos imaginarios en un mundo virtual. En UML las formas en las que se conectan entre sí las clases, lógica o físicamente, se modelan como relaciones. En el modelado orientado a objetos existen tres clases de relaciones muy importantes: *dependencias, generalizaciones-especializaciones y asociaciones* (Booch 2006):

- Las *dependencias* son relaciones de uso.
- Las *asociaciones* son relaciones estructurales entre objetos. Una relación de asociación "todo/parte", en la cual una clase representa una cosa grande ("el todo") que consta de elementos más pequeños ("las partes") se denomina *agregación.*
- Las *generalizaciones* conectan clases generales con otras más especializadas en lo que se conoce como relaciones subclase/superclase o hijo/padre.

Una *relación* es una conexión entre elementos; en el modelado orientado a objetos se representa gráficamente con una línea (continua, punteada) que une las clases. En UML la relación es uno de los bloques o componentes más importantes; muestra cómo los elementos se asocian unos con otros y esta asociación describe la funcionalidad de una aplicación. En UML 2.5.1 existen cuatro tipos de relaciones disponibles (tabla 17.1):

Tabla 17.1. Relaciones de clases.

Relación	Descripción y representación gráfica
Dependencia	Relación entre dos cosas en la que cualquier cambio en un elemento afecta también al otro. 
Asociación	Conjunto de elementos que conectan elementos de un modelo UML. También describe cuántos objetos forman parte de la relación. 
Generalización	Es una relación que conecta un elemento especializado con un elemento generalizado. Básicamente describe la relación de herencia en el mundo de objetos. 
Realización	Es una relación en la cual dos elementos están conectados. Un elemento describe una responsabilidad que no está implementada y el otro elemento la implementa. Esta relación existe en el caso de las interfaces. 

Cristian Pilapaña 1095211-2202112348

17.2. Dependencia

La relación más débil que puede existir entre dos clases es de *dependencia*; que significa que una clase utiliza o tiene conocimiento de otra; dicho de otro modo "lo que una clase necesita conocer de otra para utilizar objetos de esa clase" (Russ Miles & Kim Hamilton, *Learning UML 2.0*, O'Reilly, pp. 81-82). Por lo general, es una relación transitoria y significa que una clase dependiente interactúa brevemente con la clase destino, pero normalmente no tiene con ella una relación de un tiempo definido. Una *dependencia* es una relación de uso que declara que un elemento utiliza la información y los servicios de otro elemento pero no necesariamente a la inversa.

La dependencia se lee normalmente como una relación *"...usa un..."*; por ejemplo, si se tiene una clase `Ventana` que envía un aviso a una clase llamada `EventoCerrarVentana` cuando está próxima a abrirse, en ese caso se dice que `Ventana` utiliza `EventoCerrarVentana`.

Otro ejemplo puede ser una clase `InterfazUsuario` que depende de otra clase `EntradaBlog` ya que necesita leer el contenido de la entrada de un *blog* (página web) para visualizar al usuario.

En un diagrama de clases, la dependencia se representa utilizando una línea discontinua dirigida hacia el elemento del cual depende. La flecha punteada de dependencia (figura 17.2) significa "se utiliza simplemente cuando se necesita y se olvida luego de ella".

Figura 17.2. Relación de dependencia. `Ventana` depende de la clase `EventoCerrarVentana` porque necesitará leer el contenido de esta clase para poder cerrar la ventana.

Cristian Pilapanta 1095211-2202112348

Otro ejemplo de dependencia se muestra entre la clase `Interfaz` y la clase `EntradaBlog` ya que ambas trabajan juntas; `Interfaz` necesitará leer el contenido de las entradas del *blog* para visualizar estas entradas al usuario. Las dependencias se usarán cuando se quiera indicar que un elemento utiliza a otro. Una dependencia implica que los objetos de una clase pueden trabajar juntos; por consiguiente, se considera que es la relación directa más débil que puede existir entre dos clases

17.3. Asociación

Una **asociación** es más fuerte que la dependencia y normalmente indica que una clase recuerda o retiene una relación con otra durante un periodo determinado. Es decir, las clases se conectan juntas conceptualmente en una asociación. La asociación realmente significa que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo; y se representa utilizando una simple línea que conecta las dos clases, como se muestra en la figura 17.3.

Una *asociación* es una relación estructural que especifica que los objetos de una clase están conectados con los objetos de otra clase. En general, si se encuentra que una clase *trabaja* con un objeto de otra, entonces la relación es una buena candidata para una asociación en lugar de una dependencia.

Gráficamente, una asociación se representa como una línea continua que conecta la misma o diferentes clases. Las asociaciones se deben utilizar cuando se desee representar relaciones estructurales. Los adornos de una asociación son: línea continua, nombre de la asociación, dirección del nombre mediante una flecha que apunta en la dirección de una clase a la otra. La *navegabilidad* se aplica a una relación de asociación que describe qué clase contiene el atributo que soporta la relación.

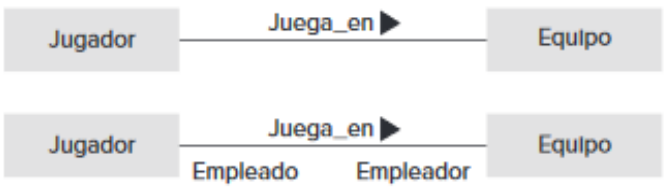


Figura 17.3. Relación de asociación (Jugador-Equipo).

En notación moderna, a esta relación suele llamársele también *tiene un*, pero hay que tener cuidado porque este concepto es sutil y de hecho siempre ha representado a la agregación, pero como se verá después UML contempla que la agregación *posee* (... *owns a*...). Por esta razón nos inclinaremos en considerar la relación *tiene-un* (*has-a*) como la relación de agregación.

La multiplicidad representa la cantidad de objetos de una clase que se relacionan con un objeto de la clase asociada. La información de multiplicidad aparece en el diagrama de clases a continuación del rol correspondiente. La multiplicidad se escribe como una expresión con un valor mínimo y un valor máximo, que pueden ser iguales; se utilizan dos puntos consecutivos para separar ambos valores. Cuando se indica una multiplicidad en un extremo de una asociación se está especificando cuántos objetos de la clase de ese extremo pueden existir por cada objeto de la clase en el otro extremo. UML utiliza un asterisco (*) para representar *más* y representa *muchos*. La tabla 17.2 resume los valores más típicos de multiplicidad.

Tabla 17.2. Multiplicidad en asociaciones.

Símbolo	Significado
1	Uno y sólo uno
0 .. 1	Cero o uno
m .. n	De m a n (enteros naturales)
*	De cero a muchos (cualquier entero positivo)
0 .. *	De cero a muchos (cualquier entero positivo)
1 .. *	De uno a muchos (cualquier entero positivo)
2	Dos
5 .. 11	Cinco a once
5, 10	Cinco o diez

Si no se especifica multiplicidad, es uno (1) por omisión. La multiplicidad se muestra cerca de los extremos de la asociación, en la clase donde es aplicable.

EJEMPLO 17.1

Relación de asociación entre las clases `Empresa` *y* `Persona`.

Cada objeto `Empresa` tiene como `empleados`, 1 o más objetos `Persona` (`Multiplicidad 1.. *`); pero cada objeto `Persona` tiene como `patrón` a cero o más objetos `Empresa`.

17.3.2. Restricciones en asociaciones

En algunas ocasiones, una asociación entre dos clases ha de seguir una regla. En este caso, la regla se indica poniendo una restricción cerca de la línea de la asociación que se representa por el nombre encerrado entre llaves.

Cristian Pilponi 1095211-2021112348

EJEMPLO 17.2

Un cajero de un banco (humano o electrónico) atiende a clientes; esto se realiza en el orden en que se colocan ante la ventanilla o mostrador, o bien en función del momento de la petición electrónica de acceso al cajero.

Figura 17.12. Restricción en una asociación.

En algunas ocasiones las asociaciones pueden establecer una restricción entre las clases; una restricción típica puede ser {ordenado} `{or}`.

17.3.3. Asociación cualificada

Cuando la multiplicidad de una asociación es de uno a muchos, se puede reducir esta multiplicidad de uno a uno con una cualificación. El símbolo que representa la cualificación es un pequeño rectángulo adjunto a la clase correspondiente.

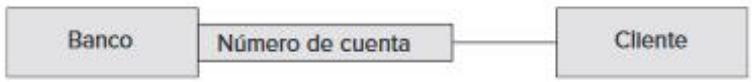


Figura 17.13. Asociación cualificada.

17.3.4. Asociaciones reflexivas

A veces, una clase es una asociación consigo misma; por ejemplo, puede suceder que una clase tiene objetos que pueden jugar diferentes roles.

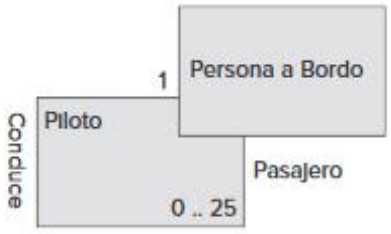


Figura 17.14. Asociación reflexiva.

17.3.5. Diagrama de objetos

Los objetos se pueden representar en diagramas de objetos. Un diagrama de objetos en UML tiene la misma notación y relaciones que un diagrama de clases, dado que los objetos son instancias de las clases. Así, un diagrama de clases muestra los tipos de clases y sus relaciones, mientras que el diagrama de objetos muestra instancias específicas de esas clases y enlaces específicos entre esas instancias en un momento dado. El diagrama de objetos mues-

Cristian Pilaparta 1095211-2202112348

tra también cómo los objetos de un diagrama de clases se pueden combinar con cada uno de los restantes en un cierto instante.

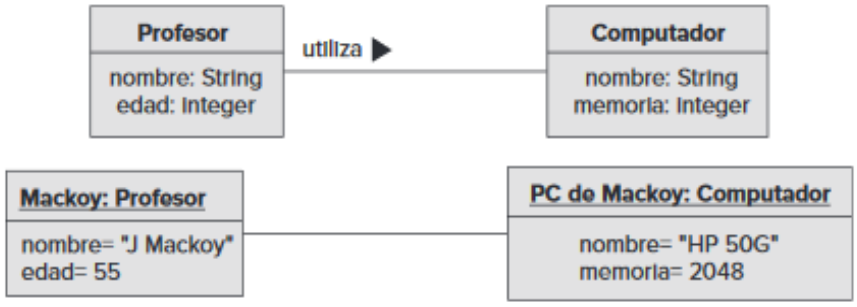


Figura 17.15. Diagrama de clases y diagrama de objetos.

Enlaces

Al igual que un objeto es una instancia de una clase, una asociación tiene también instancias; por ejemplo, la asociación de la figura 17.16.

Figura 17.16. Instancia de una asociación.

Cristian Pilaparita 1095211-2202112348

17.3.6. Clases de asociación

Es frecuente encontrarse con una asociación que introduce nuevas clases. Una clase se puede conectar a una asociación, en cuyo caso se denomina *clase asociación*; y de hecho una asociación puede tener atributos y operaciones tal como una clase, éste es el caso de la clase asociación.

La clase asociación no se conecta a ninguno de los extremos de la asociación, sino que se conecta a la asociación real, a través de una línea punteada; esta clase se utiliza para añadir información extra en un enlace, por ejemplo, el momento en que fue creado. Cada enlace de la asociación se relaciona a un objeto de la clase asociación.

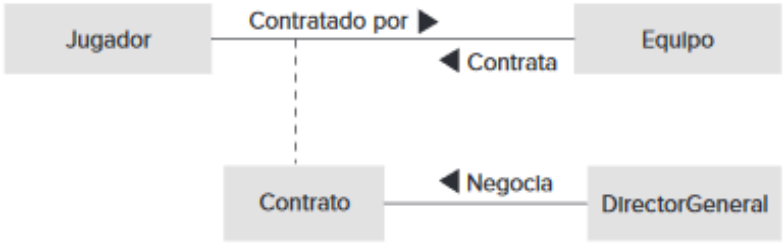


Figura 17.17. La clase asociación Contrato está asociada con la clase DirectorGeneral.

Una clase asociación es una asociación —con métodos y atributos— que es también una clase normal, y se representa con una línea punteada que la conecta a la asociación que representa.

Las clases asociación se pueden aplicar en asociaciones binarias y n-arias. De modo similar a como una clase define las características de sus objetos, incluyendo sus características estructurales y de comportamiento, una clase asociación se puede utilizar para definir las características de sus enlaces, incluyendo sus características estructurales y de comportamiento. Estos tipos de clases se utilizan cuando se necesita mantener información sobre la propia relación.

EJEMPLO 17.3

Clase asociación `EquipoFutbol`.

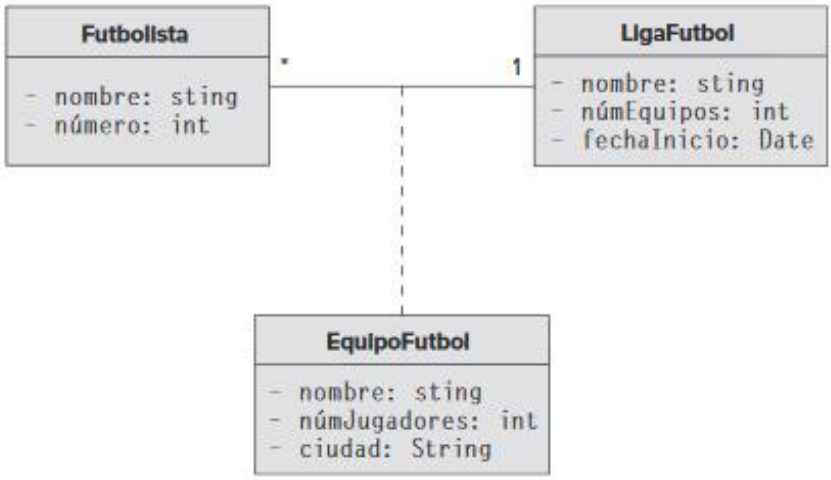


Figura 17.18. Clase asociación `EquipoFutbol`.

Cristian Pilaparta 1095211-2202112348

Criterios de diseño

Cuando se traducen a código, las relaciones con clases asociación se obtienen, normalmente, tres clases: una por cada extremo de la asociación y una por la propia clase asociación.

EJERCICIO 17.1

Diseñar el control de 5 ascensores (elevadores) de un edificio comercial que tenga presente las peticiones de viaje de los diferentes clientes, en función del orden y momento de llamada.

Análisis

El control de ascensores tiene un enlace con cada uno de los 5 ascensores y otro con el botón (pulsador) de llamada de "subida/bajada". Para gestionar el control de llamadas de modo que responda el ascensor, que cumpla con los requisitos estipulados (situado en el piso más cercano, parado, en movimiento, etcétera) se requiere una clase `Cola` que almacene las peticiones tanto del `ControlAscensor` como del propio ascensor (los motores interiores del ascensor). Cuando el control del ascensor elige uno para realizar la petición de un pasajero externo al ascensor, un pasajero situado en un determinado piso o nivel, el control del ascensor lee la cola y elige el ascensor que está situado, disponible y más próximo en la `Cola`. Esta elección normalmente se realizará por algún algoritmo inteligente.

En consecuencia, se requieren cuatro clases: `ControlAscensor`, `Ascensor` (elevador), `Botón` (pulsador) y `Cola`. La clase `Cola` será una clase asociación ya que puede ser requerida tanto por el control de ascensores como por cualquier ascensor.

Recuerde el lector que una estructura de datos cola es una estructura en la que cada elemento que se introduce en la cola es el primer elemento que sale de la cola (al igual que sucede con la cola para sacar una entrada de cine, comprar el pan o una cola de impresoras conectadas a una computadora central). En cada enlace entre los ascensores y el control de ascensores hay una cola. Cada cola almacena las peticiones del control del ascensor y el propio ascensor (los botones internos del ascensor).

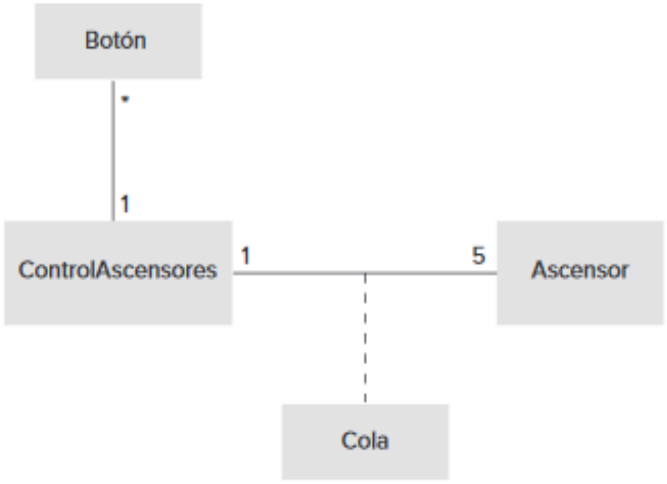


Figura 17.19. Diagrama de clases para control de ascensores.

Asociaciones ternarias

Las clases se pueden asociar una a una o se pueden asociar unas con otras. La asociación ternaria es especial debido a que asocia a tres clases, y se representa con la figura geométrica "rombo" y con los roles y multiplicidad necesarias, pero no están permitidos los cualificadores ni la agregación. Se puede conectar una clase asociación a la asociación ternaria, dibujando una línea punteada a uno de los cuatro vértices del rombo.

Cristian Pilapaña 1095211-2202112348

EJERCICIO 17.2

Dibujar un modelo de seguros de automóviles que represente: compañía de seguros, asegurados, póliza de seguro y el contrato de seguro.

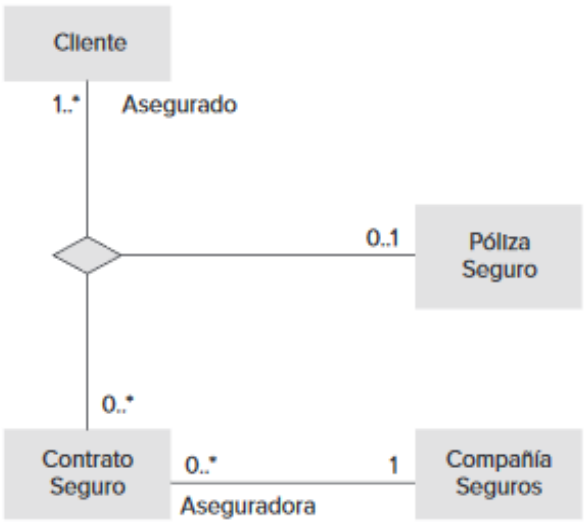


Figura 17.20. Diagrama de clases con una asociación ternaria.

El núcleo muestra un cliente –juega el rol o papel de asegurado– que puede tener 0 o muchos contratos de seguros y cada uno está asociado con una única compañía de seguros que juega el rol de aseguradora. En la asociación entre cliente y contrato de seguro, hay una o ninguna póliza de seguros.

Asociaciones cualificadas

Una asociación cualificada se utiliza con asociaciones una-a-muchas o muchas-a-muchas para reducir su multiplicidad a una, con la finalidad de especificar un objeto único (o grupos de objetos) desde el destino establecido. La asociación cualificada es muy útil para modelar cuando se busca o navega para encontrar objetos específicos en una colección determinada.

Ejemplos típicos son los sistemas de reservas de pasaje de avión, de entradas de cine, de reservas de habitaciones en hoteles. Cuando se solicita un pasaje, una entrada o una habitación, es frecuente que nos den un localizador de la reserva (H234JK, o similar) que se ha de proporcionar físicamente a la hora de sacar el pasaje en el aeropuerto, la entrada en el cine o ir a alojarse en el hotel.

El atributo o calificador se conoce normalmente como *identificador* (número de ID); existen numerosos calificadores tales como: ID de reserva, nombre, número de la tarjeta de crédito, número de pasaporte, etc. En terminología de proceso de datos, también se conoce como clave de búsqueda; este identificador o calificador, al especificar una clave única resuelve la relación uno a muchos y lo convierte en uno a uno.

El calificador se representa como una caja o rectángulo pequeño que se dibuja en el extremo correspondiente de la asociación, al lado de la clase a la cual está asociado. El calificador representa un añadido a la línea de la asociación y fuera de la clase. Las asociaciones cualificadas reducen la multiplicidad real en el modelo, de uno-a-muchos a uno-a-uno indicando con el calificador una identidad para cada asociación.

EJEMPLO 17.4

1. Lista de reservas, calificador, ID de la reserva.

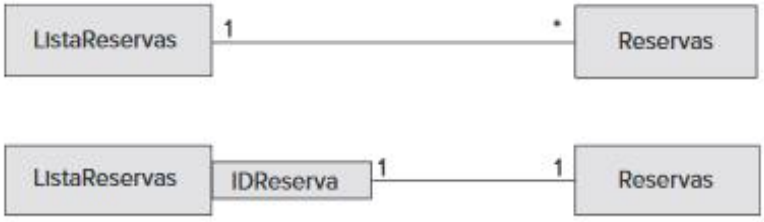


Figura 17.21. Asociación cualificada.

2. Lista de pasajes: vuelo Madrid-Cartagena de Indias con la compañía aérea Iberia.

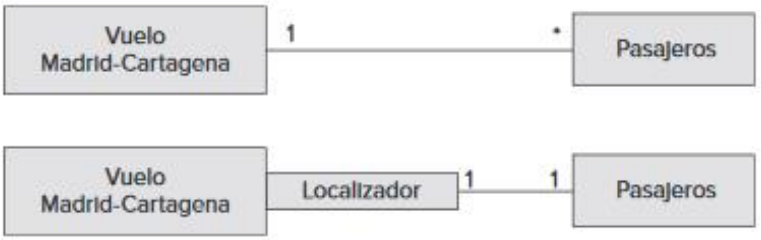


Figura 17.22. Asociación cualificada.

Asociaciones reflexivas

En ocasiones una clase es una asociación consigo misma; en este caso se denomina asociación reflexiva. Esta situación se produce cuando una clase tiene objetos que pueden jugar diferentes roles; por ejemplo, un ocupante de un avión de pasajeros puede ser: un pasajero, un miembro de tripulación o un piloto. Este tipo de asociación se representa gráficamente dibujando la línea de asociación con origen y final en la propia clase y con indicación de los roles y multiplicidades correspondientes.

Cristian Pilapaña 1095211-2202112348

EJEMPLO 17.5

1. Asociación reflexiva `OcupanteAvión`.

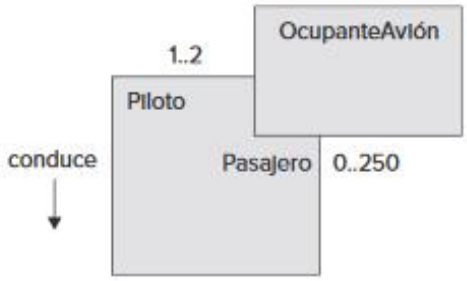


Figura 17.23. Asociación reflexiva.

2. Asociación reflexiva `OcupanteAuto`.

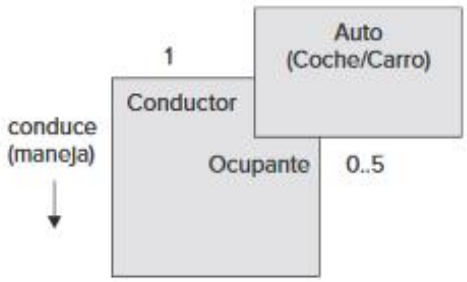


Figura 17.24. Asociación reflexiva.

Cristian Pilapaña 1095211-2202112349

17.3.7. Restricciones en asociaciones

Una asociación entre clases, a veces, tiene que seguir una regla determinada que se indica poniendo una restricción cerca de la línea de la asociación. Una restricción típica se produce cuando una clase (un objeto) atiende a otra en función de un determinado orden o secuencia; por ejemplo, un vendedor de entradas de cine (taquillero) atiende a los espectadores a medida que se sitúan delante de la ventanilla de entradas, en este caso, esta restricción se representa en el modelo con la palabra *ordered* encerrada entre llaves.

Figura 17.25. Restricciones entre asociaciones.

Figura 17.26. Asociación con una restricción. (El cajero atiende al cliente por orden de llegada a caja.)

Otro tipo de restricciones se pueden presentar y se representan con relaciones *or* o bien *xor*, y se simbolizan gráficamente con una línea de asociación y las palabras *or*, *xor* entre llaves.

EJEMPLO 17.6

Un estudiante se matricula en una universidad en estudios de ingeniería o de ciencias.

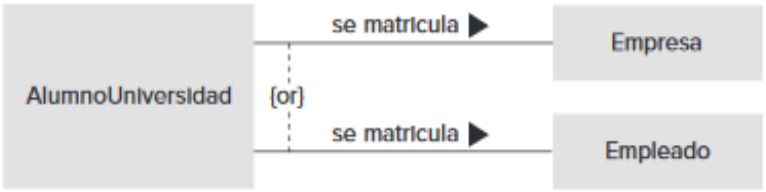


Figura 17.27. Restricción en una asociación.

EJEMPLO 17.7

La relación xor implica una u otra asociación y no pueden ser nunca las dos; el caso de una póliza de seguro de una empresa que puede ser o corporativa o de empleado, pero son entre sí, excluyentes.

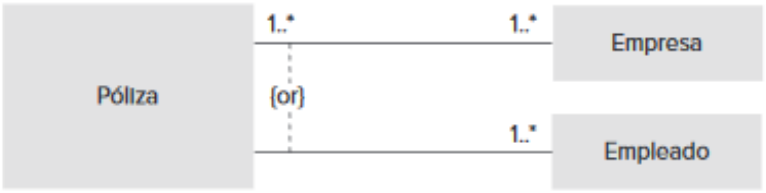


Figura 17.28. Relación xor en una asociación.

Cristian Pilapanta 1095211-2202112349

EJEMPLO 17.8

Un cajero de un banco (humano o electrónico) atiende a clientes. La atención a los clientes se realiza en el orden en que se colocan ante la ventanilla o mostrador, o bien en función del momento de la petición electrónica de acceso al cajero.

Enlaces

Al igual que un objeto es una instancia de una clase, una asociación también tiene instancia. Por ejemplo, la asociación `Juega_en` y su instancia se muestran en la figura 17.29.

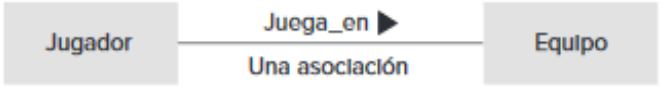


Figura 17.29. Instancia de una asociación.

17.4. Agregación

Una **agregación** es un tipo especial de asociación que expresa un acoplamiento más fuerte entre clases; una de ellas juega un papel importante dentro de la relación con las otras clases. La agregación permite la representación de relaciones tales como "maestro y esclavo", "todo y parte de" o "compuesto y componentes". Los componentes y la clase que constituyen son una asociación que conforma un todo.

Las agregaciones representan conexiones bidireccionales y asimétricas; el concepto de agregación desde un punto de vista matemático es una relación que es transitiva, asimétrica y puede ser reflexiva.

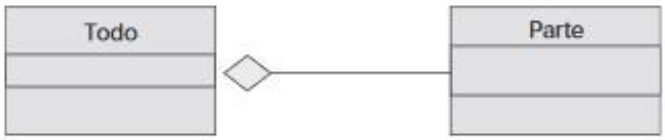


Figura 17.30. Relación de agregación.

La *agregación* es una versión más fuerte que la asociación, de hecho, normalmente implica propiedad o pertenencia; por lo general, se lee como relación ... *posee un*... o *todo-parte*, en la cual una clase ("el todo") representa un gran elemento que consta de otros más pequeños (las partes). La agregación se representa con un rombo a continuación de la clase "propietaria" y una línea recta que apunta a la clase "poseída". Esta relación se conoce como *tiene-un* ya que el todo tiene sus partes; un objeto es parte de otro objeto.

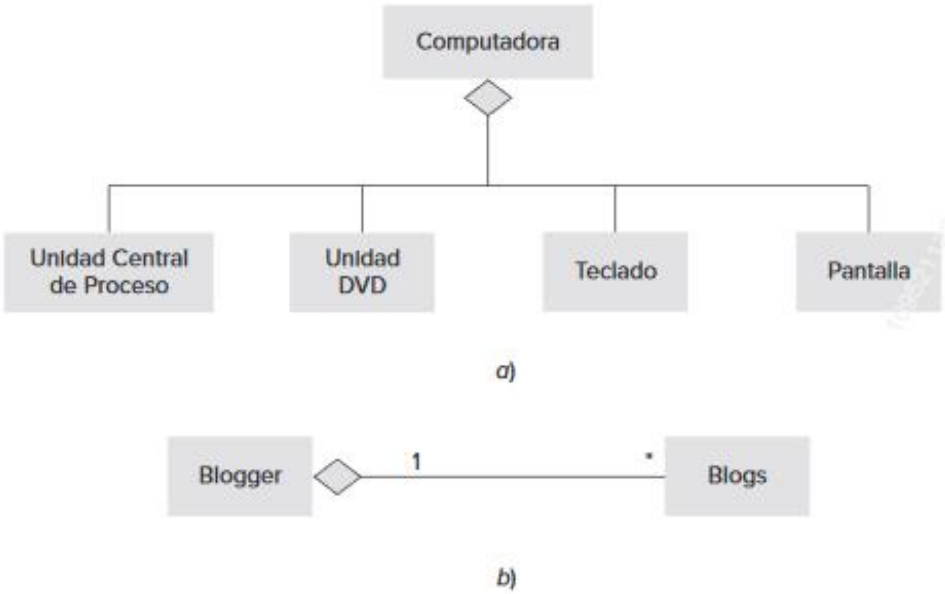


Figura 17.31. Relaciones de agregación: *a*) Computadora con sus componentes; *b*) Un *Blogger* propietario de *muchos* (*) *Blogs*.

Desde el punto de vista conceptual una clase realmente *posee*, pero puede *compartir* objetos de otra clase. Una agregación es un caso especial de asociación; un ejemplo podría ser un automóvil que consta de cuatro ruedas, un motor, un chasis, una caja de cambios, etcétera. Otro ejemplo es un árbol binario que consta de cero, uno o dos nuevos árboles. Una agregación se representa como una jerarquía con la clase "todo" (por ejemplo, un sistema de computadora) en la parte superior y sus componentes en las partes inferiores (por ejemplo CPU, discos, webcam...). La representación de la agregación se realiza insertando un rombo vacío en la parte *todo*.

EJEMPLO 17.9

Una computadora es un conjunto de elementos que consta de una unidad central, teclado, ratón, monitor, unidad de CD-ROM, módem, altavoces, escáner, entre otros.

Cristian Pilaparita 1096211-2202112349

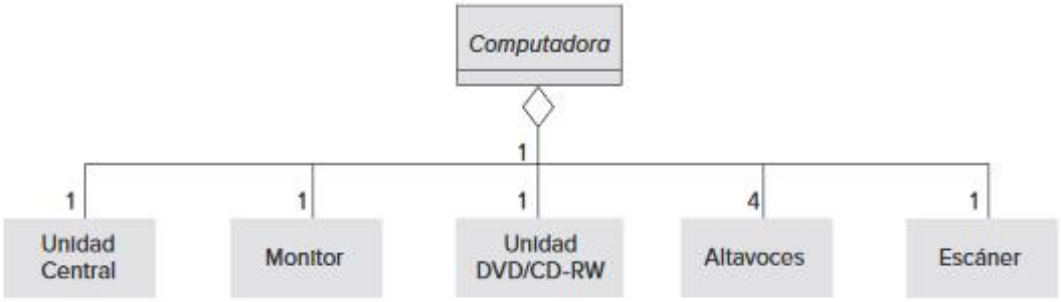


Figura 17.32. Una agregación `computadora`.

Restricciones en las agregaciones

En ocasiones, el conjunto de componentes posibles en una agregación se establece dentro de una relación *O*. Así, por ejemplo, el menú del día en un restaurante puede constar de: un primer plato (a elegir entre dos-tres platos), el segundo plato (a elegir entre dos-tres platos) y un postre (a elegir entre cuatro postres). El modelado de este tipo se realiza con la palabra reservada *O* dentro de llaves con una línea discontinua que conecte las dos líneas que conforman el todo.

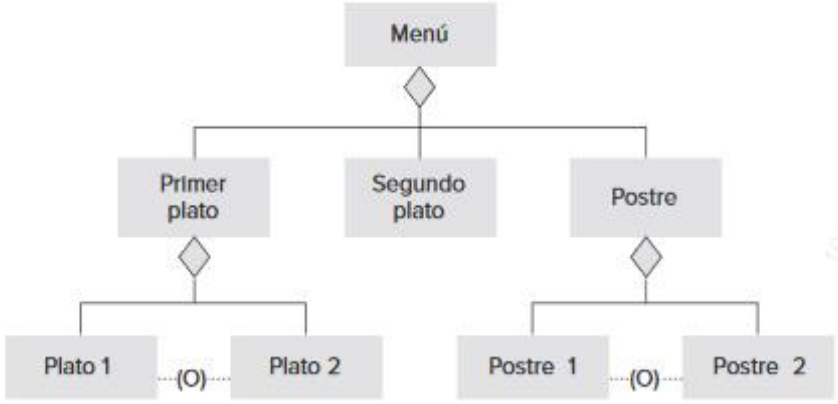


Figura 17.33. Restricción en agregaciones.

17.4.1. Composición

Una **composición** es un tipo especial de agregación que impone algunas restricciones: si el objeto completo se copia o se borra (elimina), sus partes se copian o se suprimen con él. La composición simboliza una relación fuerte entre clases y se utiliza para representar una relación *todo-parte* (*whole-part*). Cada componente dentro de una composición puede pertenecer tan sólo a un todo, el símbolo de una composición es el mismo que el de una agregación, excepto que el rombo está relleno (figura 17.34). Es como una agregación, pero con el rombo pintado y no vacío.

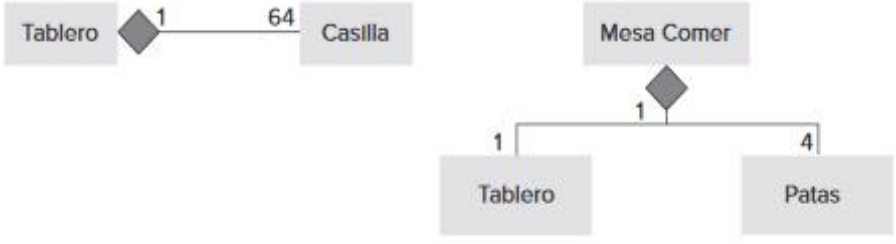


Figura 17.34. Relaciones de composición.

Cristian Pilaparta 1095211-2202112349

Una relación de composición se lee normalmente como ... es *parte de...*, que significa se necesita leer la composición de la parte al todo. Por ejemplo, si una ventana de una página web tiene una barra de títulos, se puede representar que la clase `BarraTítulo` *es parte de* una clase denominada `Ventana`.

EJEMPLO 17.10

Una mesa para jugar al póquer es una composición que consta de una superficie de la mesa y cuatro patas.

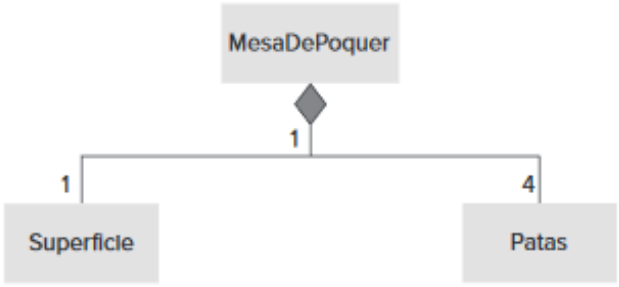


Figura 17.35. Composición.

EJEMPLO 17.11

Un auto tiene un motor que no puede ser parte de otro auto. La eliminación completa del auto supone la eliminación de su motor.

17.5. Jerarquía de clases: generalización y especialización

La jerarquía de clases (o clasificaciones) hace lo posible para gestionar la complejidad ordenando objetos dentro de árboles de clases con niveles crecientes de abstracción. Las jerarquías de clases más conocidas son: **generalización** y **especialización**.

La relación de generalización es un concepto fundamental de la programación orientada a objetos. Una *generalización* es una relación entre un elemento general (llamado *superclase* o *padre*) y un caso más concreto de ese elemento (denominado *subclase* o *hijo*). Se conoce como relación *es-un* y tiene varios nombres, *extensión*, *herencia*... Las clases modelan el hecho de que el mundo real contiene objetos con propiedades y comportamientos. La herencia modela el hecho de que estos objetos tienden a ser organizados en jerarquías, las cuales representan la relación *es-un*.

La generalización normalmente se lee como ... *es un*... comenzando en la clase específica, derivada o subclase y derivándose a la superclase o clase base. Por ejemplo, un `Gato` *es-un tipo* de `Animal`. La relación de generalización se representa con una línea continua que comienza en la subclase y termina en una flecha cerrada en la superclase.

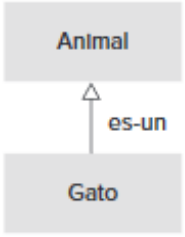


Figura 17.36. Relaciones de generalización.

Cristian Pilapanta 1095211-2202112349

En UML la relación se conoce como generalización y en programación orientada a objetos como herencia. Al contrario que las relaciones de asociación, las de generalización no tienen nombre y ningún tipo de multiplicidad.

Booch, para mostrar las semejanzas y diferencias entre clases, utiliza las siguientes clases de objetos: flores, margaritas, rosas rojas, rosas amarillas y pétalos. Se puede constatar que: una margarita *es un* tipo (una clase) de flor.

- Una rosa *es un* tipo (diferente) de flor.
- Las rosas rojas y amarillas son *tipos de* rosas.
- Un pétalo *es una parte* de ambos tipos de flores.

Como Booch afirma, las clases y objetos no pueden existir aislados y, en consecuencia, existirán entre ellos relaciones, las cuales pueden indicar alguna forma de compartición, así como algún tipo de conexión semántica. Por ejemplo, las margaritas y las rosas son tipos de flores, significa que ambas tienen pétalos coloreados brillantemente, ambas emiten fragancia, etcétera. La conexión semántica se materializa en el hecho de que las rosas rojas y las margaritas están más estrechamente relacionadas entre sí que lo que están los pétalos y las flores.

Las clases se pueden organizar en estructuras jerárquicas. La *herencia* es una relación entre clases donde una de ellas comparte la estructura o el comportamiento, definida en una (*herencia simple*) o más clases (*herencia múltiple*). Se denomina *superclase* a la clase de la cual heredan las otras. De modo similar, una clase que hereda de una o más se denomina *subclase*; una subclase heredará atributos de una superclase más elevada en el árbol jerárquico. La herencia, por consiguiente, define un "tipo" de jerarquía entre clases, en las que una subclase hereda de una o más superclases. La figura 17.37 ilustra una jerarquía de clases `Animal` con dos subclases que heredan de `Animal`, `Mamíferos` y `Reptiles`.

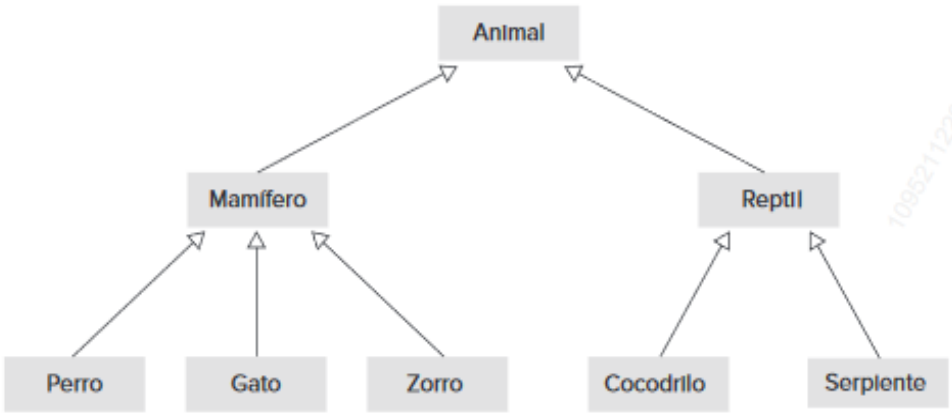


Figura 17.37. Jerarquía de clases.

Herencia es la propiedad por la cual instancias de una clase hija (o subclase) pueden acceder tanto a datos como a comportamientos (métodos) asociados con una clase padre (o superclase). La herencia siempre es transitiva, de modo que una clase puede heredar características de superclases de nivel superior. Esto es, la clase `Perro` es una subclase de la clase `Mamífero` y de `Animal`.

Una vez que una jerarquía se ha establecido es fácil extenderla. Para describir un nuevo concepto no es necesario describir todos sus atributos, basta con sus diferencias a partir de un concepto de una jerarquía existente. La herencia significa que el comportamiento y los datos asociados con las clases hija son siempre una extensión (esto es, conjunto estrictamente más grande) de las propiedades asociadas con las clases padres. Una subclase debe tener todas las propiedades de la clase padre y otras. El proceso de definir nuevos tipos y reutilizar código anteriormente desarrollado en las definiciones de la clase base se denomina *programación por herencia.* Las clases que heredan propiedades de una clase base pueden, a su vez, servir como clases base de otras clases. Esta jerarquía de tipos normalmente toma la estructura de árbol, conocido como *jerarquía de clases* o *jerarquía de tipos.*

La jerarquía de clases es un mecanismo muy eficiente, ya que se pueden utilizar definiciones de variables y métodos en más de una subclase sin duplicar sus definiciones. Por ejemplo, consideremos un sistema que representa varias clases de vehículos manejados por humanos, este sistema contendrá una clase genérica de vehículos, con subclases para todos los tipos especializados. La clase `Vehículo` contendrá los métodos y variables que fueran

Cristian Pilapaña 1095211-2202112349

propios de todos los vehículos, es decir, número de matrícula, número de pasajeros, capacidad del depósito de combustible. La subclase, a su vez, contendrá métodos y variables adicionales que serán específicos a casos individuales.

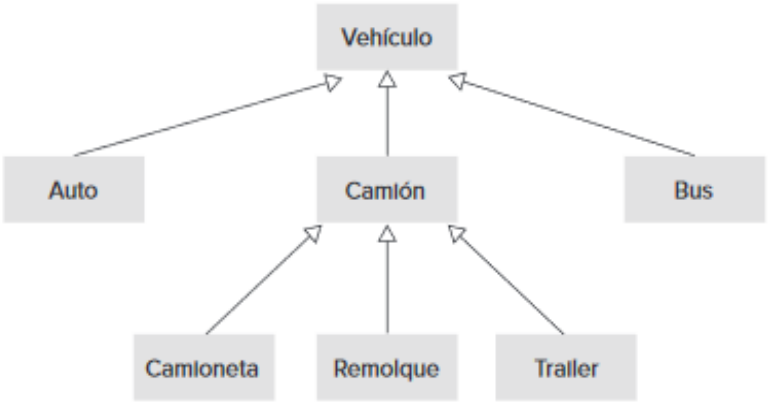


Figura 17.38. Subclases de la clase `Vehículo`.

La flexibilidad y eficiencia de la herencia no es gratuita; se emplea tiempo en buscar una jerarquía de clases para encontrar un método o variable, de modo que un programa orientado a objetos puede correr más lentamente que su correspondiente convencional. Sin embargo, los diseñadores de lenguajes han desarrollado técnicas para eliminar esta penalización en velocidad en la mayoría de los casos, permitiendo a las clases enlazar directamente con sus métodos y variables heredados, de modo que no se requiera realmente ninguna búsqueda.

Cristian Pilapaña 1095211-2202112349

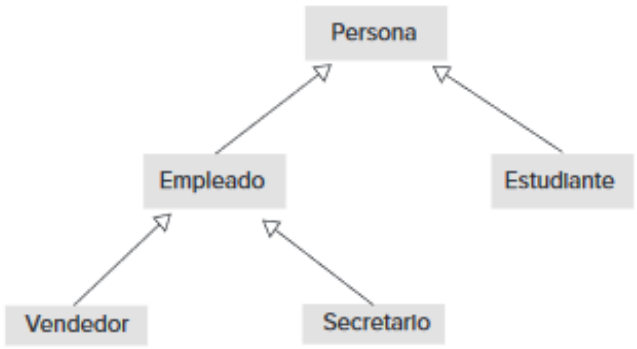


Figura 17.39. Una jerarquía `Persona`.

Regla

- Cada objeto es una instancia de una clase.
- Algunas clases —abstractas— no pueden instanciar directamente.
- Cada enlace es una instancia de una asociación.

17.5.1. Jerarquías de generalización/especialización

Las clases con propiedades comunes se organizan en **superclases**, las cuales representan una *generalización* de las subclases. De igual modo, una subclase representa una *especialización* de la clase superior (figura 17.40). La clase derivada *es-un* tipo de clase de la clase base o superclase.

Una superclase representa una *generalización* de las subclases. Una sublcase de la clase dada representa una *especialización* de la clase ascendente (figura 17.41).

En la *modelización* o *modelado* orientado a objetos es útil introducir clases en un cierto nivel que puede no existir en la realidad, pero que son construcciones conceptuales útiles. Estas clases se conocen como **clases abstractas** y su propiedad fundamental es que no se pueden crear instancias de ellas. Ejemplos de clases abstractas son `vehículo de pasajeros` y `vehículo de mercancías`. Por otra parte, de las subclases de estas clases abstractas, que corresponden a los objetos del mundo real, se pueden crear instancias directamente por sí mismas. Por ejemplo, de `BMW` se pueden obtener dos instancias, `Coche1` y `Coche2`.

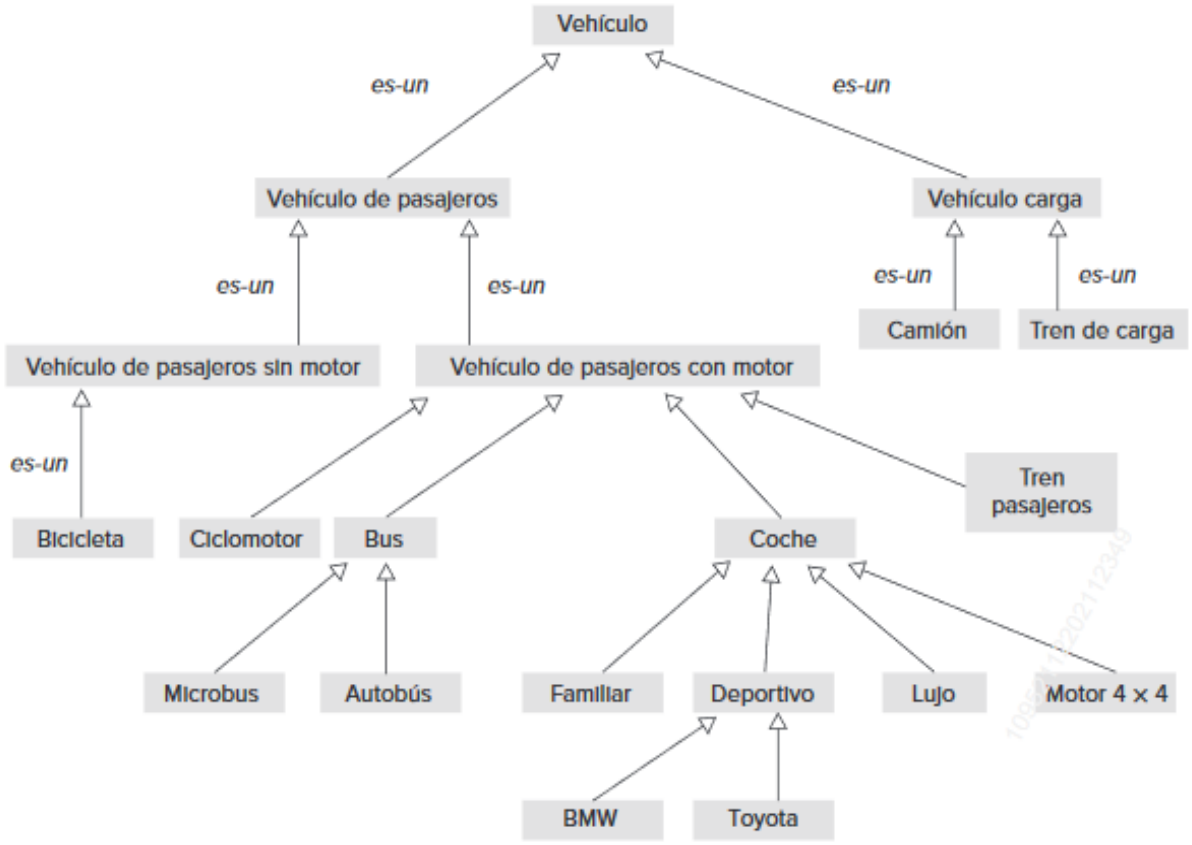


Figura 17.40. Relaciones de generalización.

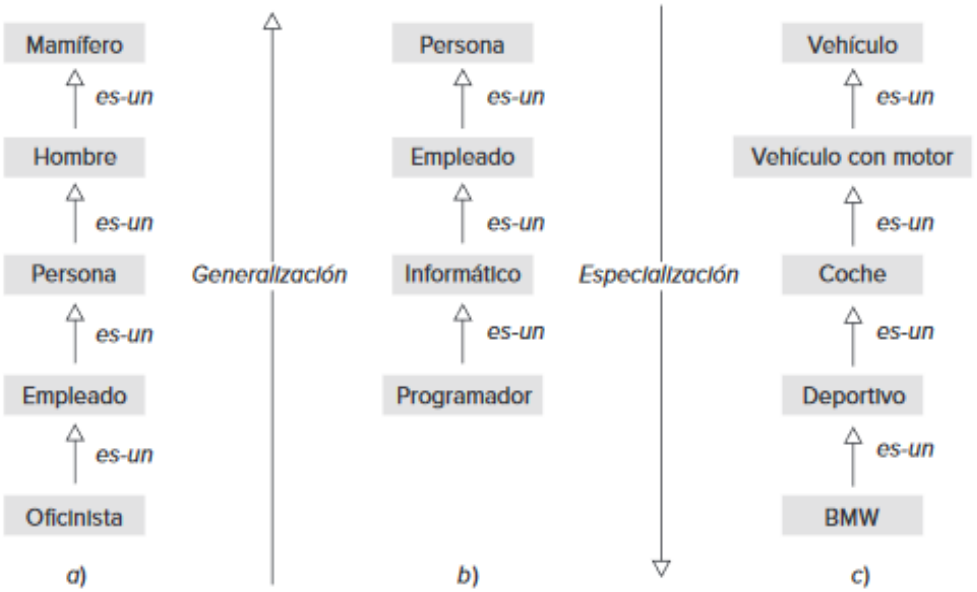


Figura 17.41. Relaciones de jerarquía *es-un (is-a)*.

Cristian Pilapanta 1095211-2202112349

La generalización, en esencia, es una abstracción, en la que un conjunto de objetos de propiedades similares se representa mediante un objeto genérico. El método usual para construir relaciones entre clases es definir generalizaciones buscando propiedades y funciones de un grupo de tipos de objetos similares, que se agrupan juntos para formar un nuevo tipo genérico. Consideremos el caso de empleados de una compañía que pueden tener propiedades comunes (nombre, número de empleado, dirección, etc.) y funciones comunes (calcular_nómina), aunque dichos empleados pueden ser muy diferentes en atención a su trabajo: oficinistas, gerentes, programadores, ingenieros, entre otros. En este caso, lo normal será crear un objeto genérico o superclase `Empleado`, que definirá una clase de empleados individuales.

Por ejemplo, `Analistas`, `Programadores` y `Operadores` se pueden generalizar en la clase `informático`. Un programador determinado (`Mortimer`) será miembro de las clases `Programador`, `Informático` y `Empleado`; sin embargo, los atributos significativos de este programador variarán de una clase a otra.

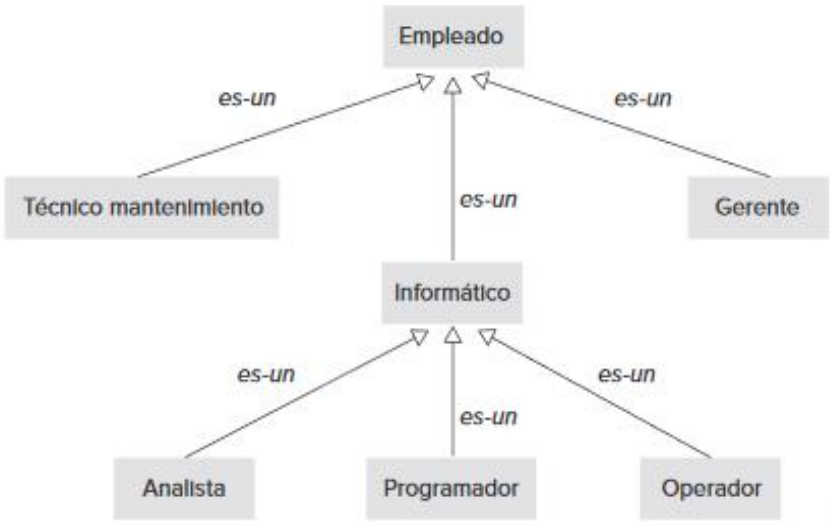


Figura 17.42. Una jerarquía de generalización de empleados.

La jerarquía de generalización/especialización tiene dos características fundamentales y notables. Primero, un tipo objeto no desciende más que de un tipo objeto genérico; segundo, los descendientes inmediatos de cualquier nodo no necesitan ser objetos de clases exclusivas mutuamente. Por ejemplo, los `Gerentes` y los `Informáticos` no tienen por qué ser exclusivos mutuamente, pero pueden ser tratados como dos objetos distintos; es el tipo de relación que se denomina *generalización múltiple.*

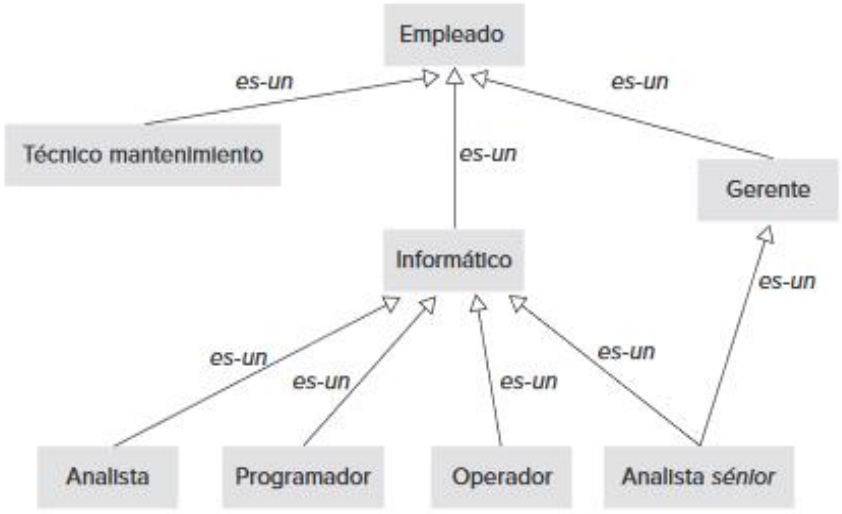


Figura 17.43. Una jerarquía de generalización múltiple.

Cristian Pilaparta 1095211-2202112349

UML define la generalización como herencia. De hecho, generalización es el concepto y herencia se considera la implementación del concepto en un lenguaje de programación.

Síntesis de Generalización/Especialización (Muller 97)

1. La generalización es una relación de herencia entre dos elementos de un modelo tal como clase. Permite a una clase heredar atributos y operaciones de otra. En realidad es la factorización de elementos comunes (atributos operaciones y restricciones) dentro de un conjunto de clases en una más general denominada **superclase**. Las clases están ordenadas dentro de una jerarquía; una superclase es una abstracción de sus subclases.
2. La flecha que representa la generalización entre dos clases apunta hacia la clase más general.
3. La especialización permite la captura de las características específicas de un conjunto de objetos que no han sido distinguidos por las clases ya identificadas. Las nuevas características se representan por una nueva clase, que es una subclase de una de las ya existentes. La especialización es una técnica muy eficiente para extender un conjunto de clases de un modo coherente.
4. La generalización y la especialización son dos puntos de vista opuestos del concepto de jerarquía de clasificación; expresan la dirección en la que se extiende la jerarquía de clases.
5. Una generalización no lleva ningún nombre específico; siempre significa "es un tipo de", "es un", "es uno de", etcétera. La generalización sólo pertenece a clases, no se puede instanciar vía enlaces y por consiguiente no soporta el concepto de multiplicidad.
6. La generalización es una relación no reflexiva: una clase no se puede derivar de sí misma.
7. La generalización es una relación asimétrica: si la clase B se deriva de la A, entonces la A no se puede derivar de la clase B.
8. La generalización es una relación transitiva: si la clase C se deriva de la B que a su vez se deriva de la A, entonces la clase C se deriva de la A.

Cristian Pilapanta 1095211-2202112349

17.6. Herencia: clases derivadas

Como ya se ha comentado, la herencia es la manifestación más clara de la relación de generalización/especialización y a la vez una de las propiedades más importantes de la orientación a objetos y posiblemente su característica más conocida y sobresaliente. Todos los lenguajes de programación orientados a objetos soportan directamente en su propio lenguaje construcciones que implementan de modo directo la relación entre clases derivadas.

La *herencia* o relación **es-un** es la relación que existe entre dos clases, en la que una denominada *derivada* se crea a partir de otra ya existente, denominada *clase base*. Este concepto nace de la necesidad de construir una nueva clase y existe una que representa un concepto más general; en este caso la nueva clase puede *heredar* de la ya existente. Así, por ejemplo, si existe una clase `Figura` y se desea crear una clase `Triángulo`, ésta puede derivarse de `Figura` ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. `Triángulo` *es-un* tipo de `Figura`. Otro ejemplo, puede ser `Programador` que *es-un* tipo de `Empleado`.

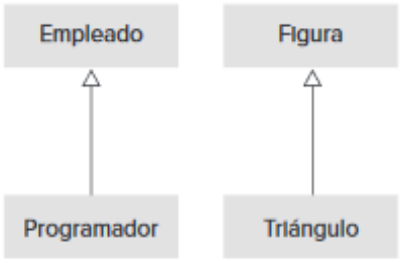


Figura 17.44. Clases derivadas.

17.6.1. Herencia simple

La implementación de la generalización es la herencia. Una clase hija o subclase puede heredar atributos y operaciones de otra clase padre o superclase., que es más general que la clase hija; a su vez, una clase hija puede ser una clase padre de otra hija. `Mamífero` es una clase derivada de `Animal` y `Caballo` es una clase hija o derivada de `Mamífero`.

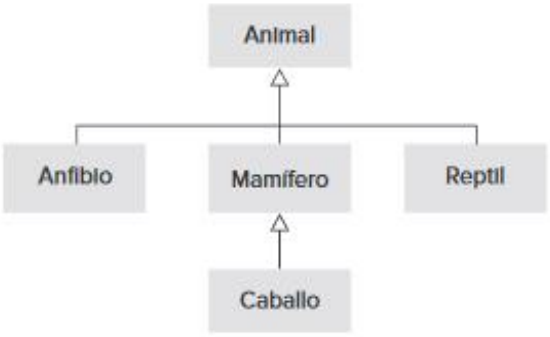


Figura 17.45. Herencia simple con dos niveles.

Cristian Pilaparia 1095211-2202112349

En UML, la herencia se representa con una línea que conecta la clase padre con la clase hija. En la parte de la línea que conecta a la clase padre se pone un triángulo abierto (punta de flecha) que apunta a dicha clase padre. Este tipo de conexión se representa como *es un tipo de*. `Caballo` *es-un-tipo* de `Mamífero` que a su vez es un tipo de `Animal`.

Una clase puede no tener padre, en cuyo caso se denomina *clase base* o *clase raíz*, y también puede no tener ninguna clase hija, en cuyo caso se le llamaría *clase terminal* o *clase hija*.

Si una clase tiene exactamente un padre, tiene **herencia simple**; si por el contrario, tiene más de un padre, tiene **herencia múltiple**.

17.6.2. Herencia múltiple

Herencia múltiple o **generalización múltiple** —en terminología oficial de UML— se produce cuando una clase hereda de dos o más clases padres (figura 17.46).

Aunque la herencia múltiple está soportada en UML y en C++ (no en Java), en general, su uso no se considera una buena práctica en la mayoría de los casos. Esta característica se debe al hecho de que la herencia múltiple presenta un problema complicado cuando las dos clases padre tienen solapamiento de atributos y comportamientos. ¿A qué se debe la complicación? Normalmente a conflictos de atributos o propiedades derivadas. Por ejemplo, si las clases `A1` y `A2` tienen el mismo atributo nombre, la clase hija `A1-2`, ¿de cuál de las dos clases hereda el atributo?

C++, que soporta herencia múltiple, debe utilizar un conjunto propio de reglas para resolver estos conflictos. Estos problemas conducen a malas prácticas de diseño y ha hecho que lenguajes de programación como **Java** y **C#** no soporten herencia múltiple. Sin embargo como C++ soporta esta característica, UML incluye en sus representaciones este tipo de herencia.

Regla

La generalización es una relación **"es-un"** (un Carro es-un Vehículo; un Gerente **es-un** Empleado, etcétera). También se utiliza **"es-un-tipo-de"** (*is a kinf of*).

En orientación a objetos la relación se conoce como *herencia* y en UML como *generalización*.

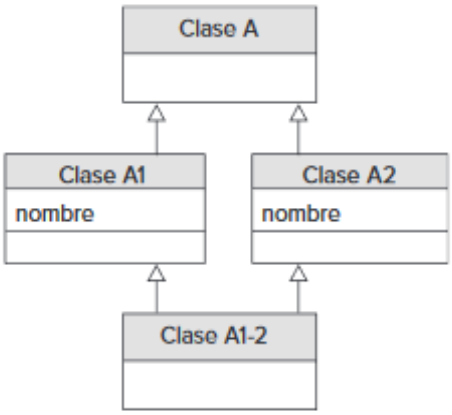


Figura 17.46. Herencia múltiple en la clase *A1-2*.

17.6.3. Niveles de herencia

La jerarquía de herencia puede tener más de dos niveles. Una clase hija puede ser una clase padre, a su vez, de otra clase hija. Así, una clase `Mamífero` es una clase hija de `Animal` y una clase padre de `Caballo`.

Las clases hija o subclases añaden sus propios atributos y operaciones a los de sus clases base. Una clase puede no tener clase hija, en cuyo caso es una *clase hija*. Si una clase tiene sólo un padre, se tiene *herencia simple* y si tiene más de un padre, entonces, se tiene *herencia múltiple*.

Cristian Pilapanta 1095211-2202112349

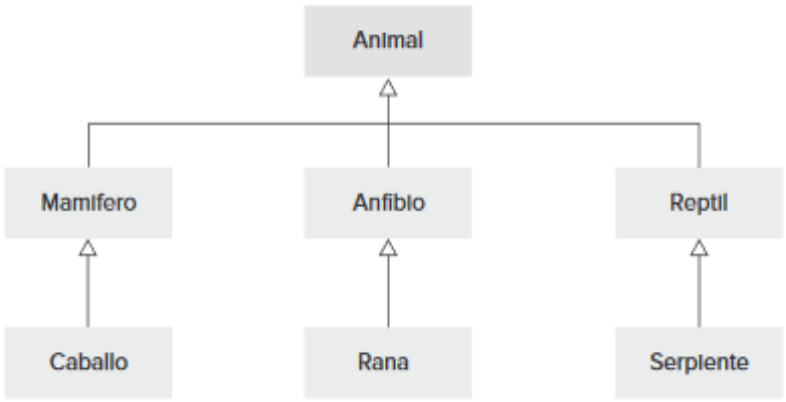


Figura 17.47. Dos niveles en una jerarquía de herencia simple.

EJEMPLO 17.12

Representaciones gráficas de la herencia.

1. `Vehículo` es una *superclase* (clase base) y tiene como clase derivadas (subclase) `Coche (Carro)`, `Barco`, `Avión` y `Camión`. Se establece la jerarquía `Vehículo`, es una *generalización-especialización*.

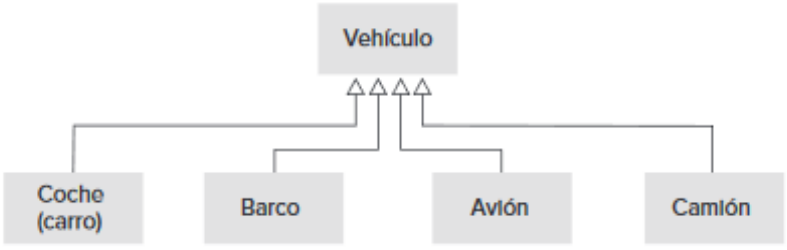


Figura 17.48. Diagrama de clases de la jerarquía *Vehículo*.

2. Jerarquía `Vehículo` (segunda representación gráfica, tipo árbol).

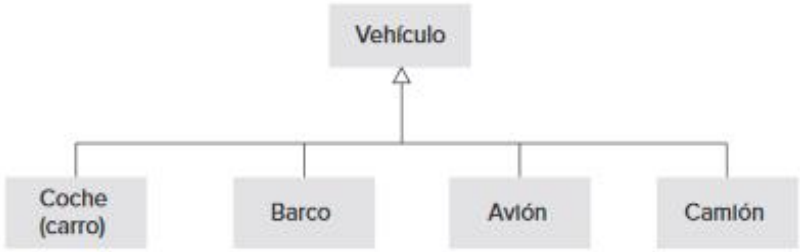


Figura 17.49. Jerarquía *Vehículo* en forma de árbol.

Evidentemente, la clase base y la clase derivada tienen código y datos comunes, de modo que si se crea la clase derivada de manera independiente, se duplicaría mucho de lo que ya se ha escrito para la clase base. C++ soporta el mecanismo de *derivación* que permite crear clases derivadas, así que la nueva clase *hereda* todos los miembros datos y las funciones miembro que pertenecen a la clase ya existente.

La declaración de derivación de clases debe incluir el nombre de la clase base de la que se deriva y el especificador de acceso que indica el tipo de herencia (*pública, privada y protegida*). La primera línea de cada declaración debe incluir el formato siguiente:

```
clase nombre_clase hereda_de tipo_herencia nombre_clase_base
```

Regla

En general, se debe incluir la palabra reservada **publica** en la primera línea de la declaración de la clase derivada, y representa herencia pública. Esta palabra reservada produce que todos los miembros que son públicos en la clase base permanecen públicos en la clase derivada.

Cristian Pilaparia 1095211-2202112349

EJEMPLO 17.13

Declaración de las clases `Programador` y `Triangulo`.

```
1. clase Programador hereda_de Empleado
   publica:
   // miembros públicos
   privada:
   // miembros privados
   fin_clase
2. clase Triángulo hereda_de Figura
      publica:
         // sección pública
      ...
      privado:
      // sección privada
   ...
```

Una vez que se ha creado una clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.

clase derivada clase base

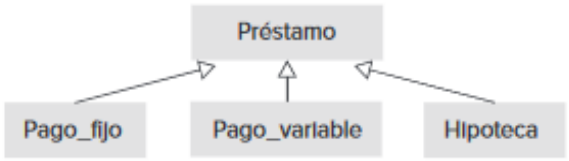
```
clase Director hereda_de Empleado
publica:
   nuevas funciones miembro
privada:
   nuevos miembros dato
fin_clase
```

En la definición de la clase `Director` sólo se especifican los miembros nuevos (funciones y datos). Todas las funciones miembro y los miembros dato de la clase `Empleado` son heredados automáticamente por la clase `Director`. Por ejemplo, la función `calcular_salario` de `Empleado` se aplica automáticamente a los directores:

```
Director d;
d.calcular_salario(325000);
```

EJEMPLO 17.14

Considere una clase Préstamo y tres clases derivadas de ella: Pago_fijo, Pago_variable e Hipoteca.



Cristian Pilapanta 1095211-2202112349

1095211220211234

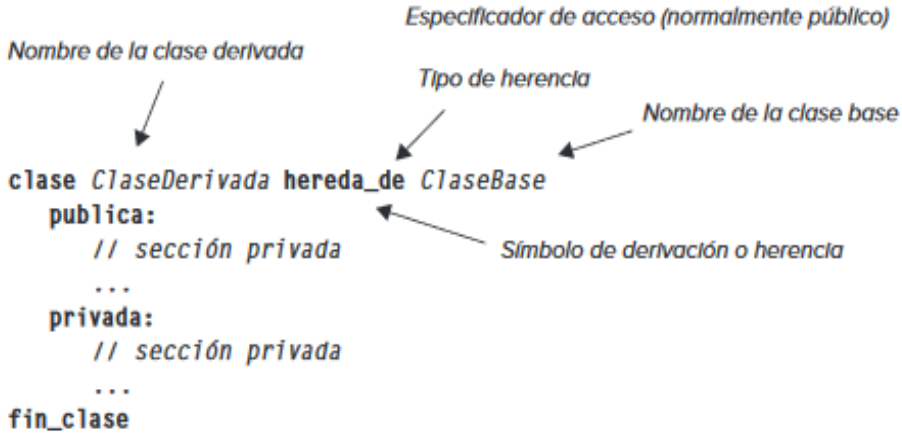
```
clase Préstamo
protegida:
   real capital;
   real tasa_interés;
publica:
   Préstamo(float, float);
   virtual int crearTablaPagos(float[MÁX_TERM][NÚM_COLUMNAS] = 0;
fin_clase
```

Las variables capital `tasa_interés` no se repiten en la clase derivada

```
clase Pago_fijo hereda_de Préstamo
privada:
   real pago;       // cantidad mensual a pagar por cliente
publica:
   Pago_Fijo (float, float, float);
   ent CrearTablaPagos(float[MÁX_TERM][NÚM_COLUMNAS]);
);

clase Hipoteca hereda_de Préstamo
privada:
   entero núm_recibos;
   entero recibos_por_anyo;
   real pago;
publica:
   Hipoteca(int, int, float, float, float);
   entero CrearTablaPagos(float [MÁX_TERN][NÚM_COLUMNAS]);
fin_clase
```

17.6.4. Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es:



Especificador de acceso `publica`, significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.

Herencia pública, es aquella en que el especificador de acceso es `publica` (*público*).
Herencia privada, es aquella en que el especificador de acceso es `privada` (*privado*).
Herencia protegida, es aquella en que el especificador de acceso es `protegida` (*protegido*).

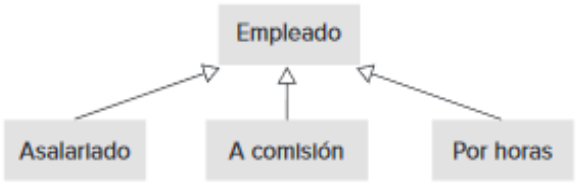
El especificador de acceso que declara el tipo de herencia es opcional (`publica`, `privada` o `protegida`); si se omite el especificador de acceso, se considera por defecto `privada`. La *clase base* (`ClaseBase`) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

```
clase nombre_clase hereda_de [especificador_acceso] ClaseBase
   lista_de_miembros;
fin_clase
```

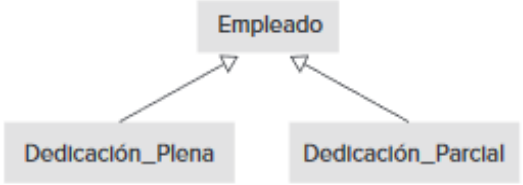
17.6.5. Consideraciones de diseño

A veces es difícil decidir cuál es la relación de herencia más óptima entre clases en el diseño de un programa. Consideremos, por ejemplo, el caso de los empleados o trabajadores de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (se suele llamar *discriminador*) y pueden ser: modo de pago (sueldo fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

Una vista de los empleados basada en el modo de pago puede dividirlos en empleados con salario mensual fijo; con pago por horas de trabajo y a comisión por las ventas realizadas.

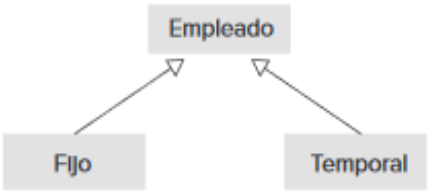


Una vista de los empleados basada en el estado de dedicación a la empresa: plena o parcial.



Cristian Pilaparda 1095211-2202112349

Una vista de empleados basada en el estado laboral del empleado con la empresa: fijo o temporal.



Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual, uno con dedicación parcial puede ser remunerado mediante comisiones y uno fijo puede ser remunerado por horas. Una pregunta usual es ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones?, ¿esta relación debe ser el fundamento del diseño de clases? Evidentemente la respuesta adecuada sólo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

17.7. Accesibilidad y visibilidad en herencia

En una clase existen secciones públicas, privadas y protegidas. Los elementos públicos son accesibles a todas las funciones; los privados son accesibles sólo a los miembros de la clase en que están definidos y los protegidos pueden ser accedidos por clases derivadas debido a la propiedad de la herencia. En correspondencia con lo anterior existen tres tipos de herencia: *pública*, *privada* y *protegida*. Normalmente el tipo de herencia más utilizada es la pública.

Con independencia del tipo de herencia, una clase derivada no puede acceder a variables y funciones privadas de su clase base. Para ocultar los detalles de la clase base y de clases y funciones externas a la jerarquía de clases, una clase base utiliza normalmente elementos protegidos en lugar de elementos privados. Suponiendo herencia pública, los elementos protegidos son accesibles a las funciones miembro de todas las clases derivadas.

Cristian Pilaparia 1095211-2202112349

Tabla 17.3. Acceso a variables y funciones según tipo de herencia.

Tipo de herencia	Tipo de elemento	¿Accesible a clase derivada?

`Publica`	`publica`
	`protegida`
	`privada`
		sí
		sí
		no

`Privada`	`publica`
	`protegida`
	`privada`
		no
		no
		no

Norma

Por defecto, la herencia es privada. Si accidentalmente se olvida la palabra reservada `publica`, los elementos de la clase base serán inaccesibles. El tipo de herencia es, por consiguiente, una de las primeras cosas que se debe verificar si un compilador devuelve un mensaje de error que indique que las variables o funciones son inaccesibles.

17.7.1. Herencia pública

En general, *herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y privados de su clase base. Los elementos públicos se heredan como elementos públicos; los elementos protegidos permanecen protegidos. La herencia pública se representa con el especificador `publica` en la derivación de clases.

Formato

```
clase ClaseDerivada hereda_de publica Clase Base
```

```
    pública:
    // sección pública
    privada:
    // sección privada
fin_clase
```

17.7.2. Herencia privada

La herencia privada significa que una clase derivada no tiene acceso a ninguno de sus elementos de la clase base. El formato es:

```
clase ClaseDerivada hereda_de privada  ClaseBase
pública:
   // sección pública
protegida:
   // sección protegida
privada:
   // sección privada
fin_clase
```

Con herencia privada, los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada. En efecto, los usuarios de la clase derivada no tienen acceso a las facilidades proporcionadas por la clase base. Los miembros privados de la clase base son inaccesibles a las funciones miembro de la clase derivada. La herencia privada se utiliza con menos frecuencia que la pública. Este tipo de herencia oculta la clase base del usuario y así es posible cambiar la implementación de la clase base o eliminarla toda junta sin requerir ningún cambio al usuario de la interfaz. Cuando un especificador de acceso no está presente en la declaración de una clase derivada, se utiliza herencia privada.

17.7.3. Herencia protegida

Con herencia protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada y los miembros privados de la clase base se vuelven inaccesibles. La herencia protegida es apropiada cuando las facilidades o aptitudes de la clase base son útiles en la implementación de la clase derivada, pero no son parte de la interfaz que el usuario de la clase ve. La herencia protegida es todavía menos frecuente que la privada.

La tabla 17.4 resume los efectos de los tres tipos de herencia en la accesibilidad de los miembros de la clase derivada. La entrada *inaccesible* indica que la clase derivada no tiene acceso al miembro de la clase base.

Tabla 17.4. Tipos de herencia y accesos que permiten.

Tipo de herencia	Acceso a miembro clase base	Acceso a miembro clase derivada
publica	publica protegida privada	publica protegida *inaccesible*
protegida	publica protegida privada	protegida protegida *inaccesible*
privada	publica protegida privada	privada privada *inaccesible*

Cristian Pilapanta 1095211-2202112349

EJERCICIO 17.3

Declarar una clase base (Base) y tres clases derivadas de ella, D1, D2 y D3

```
clase Base {
   publica:
      entero i1;
   protegida:
      entero i2;
   privada:
      entero i3;
};
clase D1: privada Base {
   nada f();
};
clase D2: protegida Base {
   nada g();
};
clase D3: publica Base {
   nada h();
};
```

Ninguna de las subclases tiene acceso al miembro i3 de la clase Base. Las tres clases pueden acceder a los miembros i1 e i2. En la definición de la función miembro f() se tiene:

```
void D1::f() {
   i1 = 0;          // Correcto
   i2 = 0;          // Correcto
   i3 = 0;          // Error
};
```

17.8. Un caso de estudio especial: herencia múltiple

Herencia múltiple es un tipo de herencia en la que una clase hereda el estado (estructura) y el comportamiento de más de una clase base. En otras palabras, hay herencia múltiple cuando una clase hereda de más de una clase; es decir, existen múltiples clases base (*ascendientes* o *padres*) para la clase derivada (*descendiente* o *hija*). Algunos de los lenguajes de programación que soportan herencia múltiple son: C++, Python, Eiffel y Perl. *Java no permite la herencia múltiple, pero se puede conseguir la implementación de la herencia múltiple usando interfaces. Ruby tampoco soporta la herencia múltiple.*

La herencia múltiple entraña un concepto más complicado que la herencia simple, no sólo con respecto a la sintaxis, sino también al diseño e implementación del compilador. La herencia múltiple también aumenta las operaciones auxiliares y complementarias y produce ambigüedades potenciales. Además, el diseño con clases derivadas por derivación múltiple tiende a producir más clases que el diseño con herencia simple. Sin embargo, y pese a los inconvenientes y ser un tema controvertido, la herencia múltiple puede simplificar los programas y proporcionar soluciones para resolver problemas difíciles. En la figura 17.50 se muestran diferentes ejemplos de herencia múltiple.

Cristian Pilaparia 1095211-2202112349

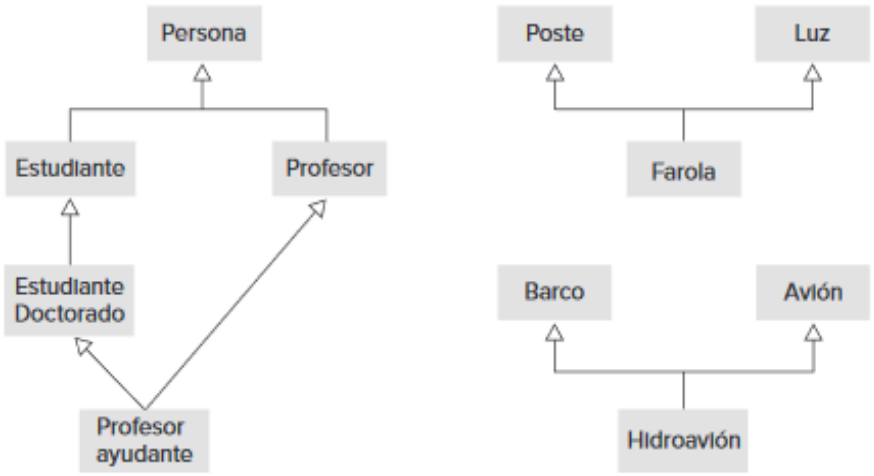


Figura 17.50. Ejemplos de herencia múltiple.

Regla

En herencia simple, una clase derivada hereda exactamente de una clase base (tiene sólo un padre). Herencia múltiple implica múltiples clases base (tiene varios padres una clase derivada).

Cristian Pilapanta 1095211-2202112349

En herencia simple, el escenario es bastante sencillo, en términos de concepto e implementación. En herencia múltiple los escenarios varían ya que las clases base pueden proceder de diferentes sistemas y se requiere a la hora de la implementación un compilador de un lenguaje que soporte dicho tipo de herencia (C++ o Eiffel). ¿Por qué utilizar herencia múltiple? Pensamos que la herencia múltiple añade fortaleza a los programas y si se tiene precaución en la base del análisis y posterior diseño, ayuda bastante a la resolución de muchos problemas que tomen naturaleza de herencia múltiple.

Por otra parte, la herencia múltiple siempre se puede eliminar y convertir en herencia simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real. La sintaxis de la herencia múltiple es:

```
clase CDerivada hereda_de Base1, Base2,...

publica:
   // sección pública
privada:
   // sección privada
...
fin_clase
```

CDerivada	Nombre de la clase derivada
Base1, Base2,...	Clases base con nombres diferentes

Funciones o datos miembro que tengan el mismo nombre en Base1, Base2, Basen, . . . serán motivo de ambigüedad.

```
clase A hereda_de publica B, C {...}
clase D hereda_de publica E, publica F, publica G {...}
```

La palabra reservada publica ya se ha comentado anteriormente, define la relación ***es-un*** y crea un subtipo para herencia simple. Así en los ejemplos anteriores, la clase A ***es-un*** tipo de B y ***es-un*** tipo de C. La clase D se deriva públicamente de E y G y privadamente de F. Esta derivación hace a D un subtipo de E y G pero no un subtipo de F. *El tipo de acceso sólo se aplica a una clase base.*

```
clase Derivada hereda_de publica Base1, Base2 {...};
```

`Derivada` especifica derivación pública de `Base1` y derivación privada (por defecto u omisión) de `Base2`.

Regla

Asegúrese de especificar un tipo de acceso en todas las clases base para evitar el acceso privado por omisión. Utilice explícitamente `privada` cuando lo necesite para manejar la legibilidad.

```
Class Derivada: publica Base1, privada Base2 {...}
```

EJEMPLO 17.15

```
clase Estudiante {
...
};
clase Trabajador {
...
};
clase Estudiante_Trabajador: publica Estudiante, publica Trabajador {
...
};
```

Cristian Pilaparita 1095211-2202112350

17.8.1. Características de la herencia múltiple

La herencia múltiple plantea diferentes problemas tales como la *ambigüedad* por el uso de nombres idénticos en diferentes clases base, y la *dominación* o *preponderancia* de funciones o datos.

Ambigüedades

Al contrario que la herencia simple, la herencia múltiple tiene el problema potencial de las ambigüedades.

EJEMPLO 17.16

```
clase Ventana {
privada:
   ...
publica:
   nada dimensionar();   // dimensiona una ventana
   ...
fin_clase

clase Fuente {
privada:
   ...
publica:
   nada dimensionar();   // dimensiona un tipo fuente
   ...
fin_clase
```

Una clase `Ventana` tiene una función `dimensionar()` que cambia el tamaño de la ventana; de modo similar, una clase `Fuente` modifica los objetos `Fuente` con `dimensionar()`. Si se crea una clase *`Ventana_Fuente`* (`VFuente`) con herencia múltiple, se puede producir ambigüedad en el uso de `dimensionar()`

```
clase VFuente: publica Ventana, publica Fuente {...};
VFuente v;
v.dimensionar();  // se produce un error ¿cuál?
```

La llamada a `dimensionar` es ambigua, ya que el compilador no sabrá a qué función `dimensionar` ha de llamar. Esta ambigüedad se resuelve fácilmente con el operador de resolución de ámbito (::)

```
v.Fuente::dimensionar();        // llamada a dimensionar() de Fuente
v.Ventana::dimensionar();       // llamada a dimensionar de Ventana
```

Precaución

No es un error definir un objeto derivado con multiplicidad con ambigüedades. Éstas se consideran ambigüedades potenciales y sólo produce errores en tiempo de compilación cuando se llaman de modo ambiguo.

Regla

Incluso es mejor solución que la citada anteriormente resolver la ambigüedad en las propias definiciones de la función dimensionar( )

```
clase VFuente: publica Ventana, publica Fuente
...
void v_dimensionar() { Ventana::dimensionar(); }
void f_dimensionar() { Fuente::dimensionar(); }
fin_clase
```

EJEMPLO 17.17

Diseñar e implementar una jerarquía de clases que represente las relaciones entre las clases siguientes: estudiante, empleado, empleado asalariado y un estudiante de doctorado que es a su vez profesor de prácticas de laboratorio.

Nota

Se deja la resolución como ejercicio al lector.

Cristian Pilaparita 1095211-2202112350

17.9. Clases abstractas

Una clase abstracta es una clase que no tiene ningún objeto; o con mayor precisión, es una clase que no puede tener objetos instancias de la clase base. Una clase abstracta describe atributos y comportamientos comunes a otras clases, y deja algunos aspectos del funcionamiento de la clase a las subclases concretas. Una clase abstracta se representa con su nombre en cursiva.

EJERCICIO 17.4

Clase abstracta `Vehículo` *con clases derivadas* `Coche(Carro)` *y* `Barco`.

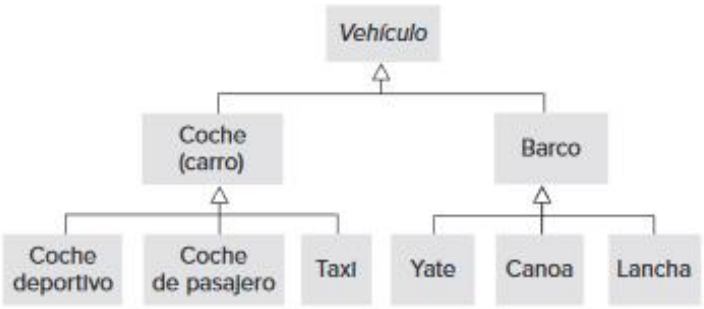


Figura 17.51. Jerarquía con clase base abstracta *Vehículo*.

Una clase abstracta se representa poniendo su nombre en cursiva o añadiendo la palabra {abstract} dentro del compartimento de la clase y debajo del nombre de la clase.

EJERCICIO 17.5

Clase abstracta `Futbolista` *de la cual derivan las clases concretas* `Portero`, `Defensa` *y* `Delantero`.

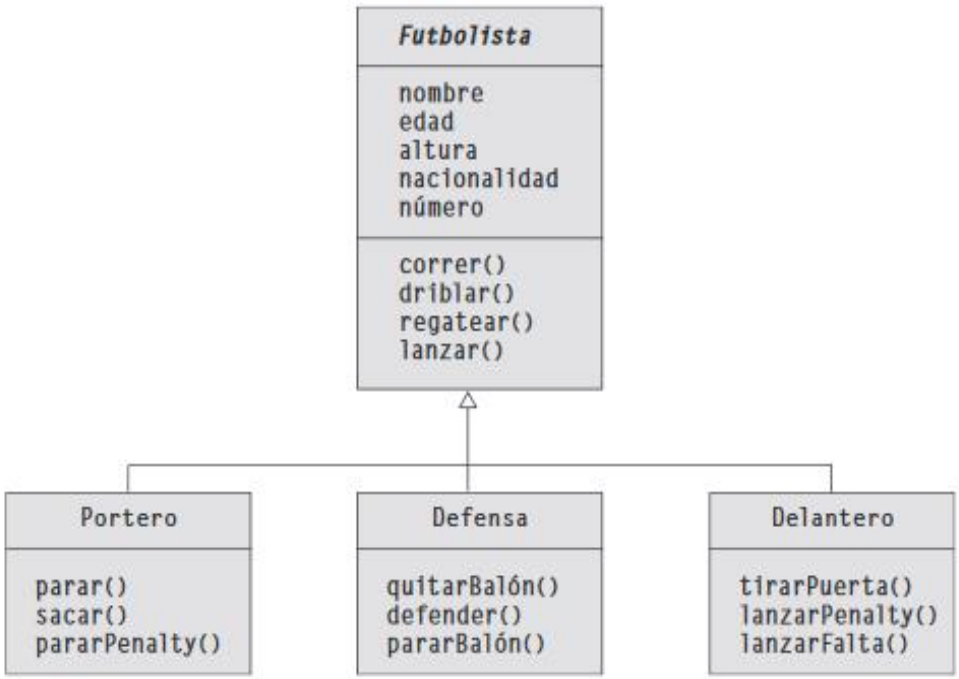


Figura 17.52. Jerarquía con clase base abstracta *Futbolista*.

Cristian Pilapanta 1095211-2202112350

CONCEPTOS CLAVE

- Agregación
- Asociación
- Clase abstracta
- Clase base
- Clase derivada
- Composición
- Constructor
- Declaración de acceso
- Destructor
- Especificadores de acceso
- Función virtual
- Generalización
- Herencia
- Herencia múltiple
- Herencia protegida
- Herencia pública y privada
- Herencia simple
- Ligadura dinámica
- Ligadura estática
- Multiplicidad
- Polimorfismo
- Relación *es-un*
- Relación *todo-parte*

RESUMEN

Una asociación es una conexión semántica entre clases; permite que una clase conozca de los atributos y operaciones públicas de otra.

Una agregación es una relación más fuerte que una asociación y representa una clase que se compone de otras. Una agregación representa la relación todo-parte; es decir una clase es el todo y contiene a todas las partes.

Una generalización es una relación de herencia entre dos elementos de un modelo tal como clases. Permite a una clase heredar atributos y operaciones de otra. Su implementación en un lenguaje orientado a objetos es la herencia. La especialización es la relación opuesta a la generalización.

La relación **es-un** representa la herencia; por ejemplo, una rosa es un tipo de flor; un pastor alemán es un tipo de perro, etc. La relación *es-un* es transitiva. Un pastor alemán es un tipo de perro y un perro es un tipo de mamífero; por consiguiente, un pastor alemán es un mamífero. Una clase nueva que se crea a partir de una ya existente, utilizando herencia, se denomina clase derivada o subclase. La clase padre se denomina clase base o superclase.

1. *Herencia* es la capacidad de derivar una clase de otra. La clase inicial utilizada por la derivada se conoce como *clase base, padre* o *superclase*. La clase derivada se conoce como *derivada, hija* o *subclase*.
2. *Herencia simple* es la relación entre clases que se produce cuando una nueva clase se crea utilizando las propiedades de una ya existente. Las relaciones de herencia reducen código redundante en programas. Uno de los requisitos para que un lenguaje sea considerado orientado a objetos es que soporte herencia.
3. La *herencia múltiple* se produce cuando una clase se deriva de dos o más clases base. Aunque es una herramienta potente, puede crear problemas, especialmente de colisión o conflicto de nombres, cosa que se produce cuando nombres idénticos aparecen en más de una clase base.

EJERCICIOS

17.1. Definir una clase base `Persona` que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.). Diseñar una jerarquía de clases que contemple las siguientes: `Estudiante`, `Empleado`, `Estudiante_empleado`. Escribir un programa que lea un archivo de información y cree una lista de personas: *a*) general; *b*) estudiantes; *c*) empleados; *d*) estudiantes empleados. El programa debe permitir ordenar alfabéticamente por el primer apellido.

Cristian Pilapaña 1095211-2202112350

17.2. Implementar una jerarquía `Libreria` que tenga al menos una docena de clases. Considere una *librería* que tenga colecciones de libros de literatura, humanidades, tecnología, etcétera.

17.3. Diseñar una jerarquía de clases que utilice como clase base o raíz una clase LAN (red de área local). Las subclases derivadas deben representar diferentes topologías, como *estrella*, *anillo*, *bus* y *hub*. Los miembros datos deben representar propiedades tales como *soporte de transmisión, control de acceso, formato del marco de datos, estándares, velocidad de transmisión*, etcétera. **Se desea simular la actividad de los nodos de tal LAN.**

La red consta de **nodos**, que pueden ser dispositivos tales como computadoras personales, estaciones de trabajo, máquinas FAX, entre otros. Una tarea principal de LAN es soportar comunicaciones de datos entre sus nodos. El usuario del proceso de simulación debe, como mínimo, poder:

- Enumerar los nodos actuales de la red LAN.
- Añadir un nuevo nodo a la red LAN.
- Quitar un nodo de la red LAN.
- Configurar la red, proporcionándole una topología de *estrella* o en *bus*.
- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro.
- Enviar un paquete de un nodo especificado a otro.
- Difundir un paquete desde un nodo a todos los demás de la red.
- Realizar estadísticas de la LAN, tales como tiempo medio que emplea un paquete.

17.4. Implementar una jerarquía. `Empleado` de cualquier tipo de empresa que le sea familiar. La jerarquía debe tener al menos cuatro niveles, con herencia de miembros dato, y métodos. Los métodos deben poder calcular salarios, despidos, promoción, dar de alta, jubilación, etc. Los métodos deben permitir también calcular aumentos salariales y primas para `Empleado` de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a `Empleado`. Por ejemplo, el tipo de acceso garantizado al público diferirá del tipo de acceso proporcionado a un supervisor de empleado, al departamento de nóminas, o al Ministerio de Hacienda. Utilice la herencia para distinguir entre al menos cuatro tipos diferentes de acceso a la información de `Empleado`.

17.5. Implementar una clase `Automovil` (*Carro*) dentro de una jerarquía de herencia múltiple. Considere que, además de ser un *Vehículo*, un automóvil es también una *comodidad, un símbolo de estado social, un modo de transporte*, etcétera. `Automovil` debe tener al menos tres clases base y al menos tres clases derivadas.

17.6. Escribir una clase `FigGeometrica` que represente figuras geométricas tales como *punto, línea, rectángulo, triángulo* y similares. Debe proporcionar métodos que permitan dibujar, ampliar, mover y destruir tales objetos. La jerarquía debe constar al menos de una docena de clases.

17.7. Implementar una jerarquía de tipos de datos numéricos que extienda los tipos de datos fundamentales como `int` y `float`, disponibles en `C++`. Las clases a diseñar pueden ser `Complejo`, `Fracción`, `Vector`, `Matriz`, etcétera.

17.8. Implementar una jerarquía de herencia de animales tal que contenga al menos seis niveles de derivación y doce clases.

17.9. Diseñar la siguiente jerarquía de clases:

Persona
Nombre
edad
visualizar( )

Estudiante		*Profesor*	
nombre	heredado	nombre	heredado
edad	heredado	edad	heredado
id	definido	salario	definido
visualizar( )	*redefinido*	visualizar( )	*heredada*

Escribir un programa que manipule la jerarquía de clases, lea un objeto de cada clase y lo visualice.

17.10. Crear una clase base denominada `Punto` que conste de las coordenadas `x` e `y`. A partir de esta clase, definir una clase denominada `Circulo` que tenga las coordenadas del centro y un atributo denominado radio. Entre las funciones miembro de la primera clase, deberá existir una función `distancia()` que devuelva la distancia entre dos puntos, donde:

$$\text{Distancia} = ((x2 - x1)^2 + (y2 - y1)^2)^{1/2}$$

17.11. Utilizando la clase construida en el ejercicio 17.10 obtener una clase derivada Cilindro derivada de `Circulo`. La clase `Cilindro` deberá tener una función miembro que calcule la superficie de dicho cilindro. La fórmula que calcula la superfi-

Cristian Pilapanta 1095211-2202112350

cie del cilindro es $S = 2r(l + r)$ donde r es el radio del cilindro y l es la longitud.

17.12. Crear una clase base denominada `Rectangulo` que contenga como miembros datos, longitud y anchura. De esta clase, derivar una clase denominada `Caja` que tenga un miembro adicional denominado profundidad y otra función miembro que permita calcular su volumen.

17.13. Dibujar un diagrama de objetos que represente la estructura de un coche (carro). Indicar las posibles relaciones de asociación, generalización y agregación.

Cristian Pilaparta 1095211-2202112350

PARTE IV

METODOLOGÍA DE LA PROGRAMACIÓN Y DESARROLLO DE SOFTWARE

Cristian Pilaparita 1095211-2202112350

CONTENIDO

CAPÍTULO 18 Ingeniería de software y metodología de la programación

CAPÍTULO 18

Ingeniería de software y metodología de la programación

18.1. Introducción a la ingeniería de software: técnicas y modelos
18.2. Proceso de desarrollo de software
18.3. Ciclo de vida del desarrollo de software
18.4. Modelos de procesos de desarrollo de software
18.5. Ciclo de vida del software "en cascada" (*waterfall*)
18.6. Metodología de la programación: análisis de especificaciones y requerimientos.
18.7. Diseño
18.8. Implementación (codificación)
18.9. Pruebas e integración
18.10. Depuración de programas
18.11. Mantenimiento
18.12. Documentación
18.13. Depuración
18.14. Diseño de algoritmos
18.15. Eficiencia
18.16. Portabilidad
CONCEPTOS CLAVE
RESUMEN

Cristian Pilaparita 1095211-2202112351

1095211220211235

INTRODUCCIÓN

La ingeniería de (del) software trata de la creación y producción de programas a gran escala. Esta actividad profesional realizada por ingenieros requiere un proceso de desarrollo del software y del trabajo de programadores de aplicaciones y de sistemas. El proceso de desarrollo de software tiene una serie de etapas que conforman un ciclo de vida y que pueden componerse de diferentes modelos. El ciclo de vida de software tradicional se conoce como "en cascada" (*waterfall*) y tiene tres modelos que se analizarán en el capítulo: Sommerville, Pressman y el clásico conocido, precisamente, como tradicional. La producción de un programa se divide en diferentes etapas: análisis, diseño, codificación, depuración, pruebas, mantenimiento y despliegue.

18.1. Introducción a la ingeniería de software: técnicas y modelos

La ingeniería de software es esencial en el funcionamiento de los gobiernos, sociedad y los negocios e instituciones nacionales e internacionales. No se puede hoy funcionar en un mundo sin software; existen diversos tipos de software, desde sistemas embebidos a sistemas de información complejos.

Muchas personas escriben programas, en las empresas se utilizan hojas de cálculo y documentos de texto para simplificar trabajos, hay quienes aprenden a programar por afición y otros de manera profesional. Sin embargo, el desarrollo de software es una actividad profesional en la que se genera software para propósitos de negocio, inclusión en otros dispositivos, o como productos, es decir, sistemas de información y sistemas de diseño asistido por computadora. La diferencia clave entre el software creado por afición y el profesional radica en que el profesional está concebido para ser utilizado por cualquier persona y no sólo por el desarrollador. El software se mantiene, cambia y evoluciona durante toda su vida.

La ingeniería de software está pensada para el soporte de desarrollo de software profesional y no para la programación. Incluye técnica que soportan especificaciones, diseño y evolución de programas, ninguna de las cuales es normalmente relevante al desarrollo de software personal.

Sommerville (p. 2016: 20) en su obra de referencia de ingeniería de software describe y explica:

- ¿Qué es software?
- ¿Cuáles son los atributos de un buen software?
- ¿Cuáles son las actividades fundamentales de la ingeniería de software?

Los costos, los retos a que se enfrentan la ingeniería de software, la diferencia entre ingeniería de software e ingeniería de sistemas, así como entre ingeniería de software y ciencia de las computadoras. Algunas de estas respuestas son:

¿Qué es software?

Programas de computadoras y documentación asociada. Los productos de software se pueden desarrollar para un cliente en particular o para un mercado general.

Un buen software se debe entregar con la funcionalidad requerida y rendimiento (desempeño) al usuario y debe ser mantenible, confiable y utilizable.

¿Qué es ingeniería de software?

Es una disciplina de ingeniería que comprende todos los aspectos de la producción de software, desde su concepción inicial hasta su funcionamiento (operación) y mantenimiento.

¿Cuáles son las actividades fundamentales de la ingeniería de software?

- Especificaciones del software.
- Desarrollo del software.
- Validación del software.
- Evolución del software.

Diferencias entre ingeniería de sistemas/informática e ingeniería de software

La ciencia de computadoras (informática) se centra en la teoría y fundamentos mientras que la ingeniería de software se centra en las prácticas de desarrollo y entrega de software útil. La ingeniería de sistemas se ocupa de todos los aspectos de la informática (sistemas basados en computadoras) incluyendo hardware, software e ingeniería de procesos. La ingeniería de software es parte de este proceso más general.

Muchas personas piensan que el software es simplemente otro término para definir los programas de computadora. Sin embargo, Sommerville (p. 19) menciona que la ingeniería de software[1] comprende todos los aspectos de la producción del software; por lo tanto, el software no representa sólo el programa, sino también todo lo asociado a documentación, librerías (biblioteca), sitios web de soporte y configuración de datos que son necesarios para

[1] El término ingeniería de software se acuñó en 1968 con ocasión de la conocida como crisis del software: `software-engineering-book.com/web/history`.

Cristian Pilapaña 1095211-2202112351

hacer estos programas útiles. Un sistema de software desarrollado profesionalmente es más que un simple programa; puede constar de diferentes programas independientes y archivos de configuración. Es posible incluir documentación de usuario que describe la estructura del sistema, cómo utilizarlo y además sitios web donde los usuarios pueden descargar información actualizada.

Los productos de software son desarrollados por los ingenieros y distribuidos o comercializados a clientes; existen dos tipos de productos de software (Sommerville 2016: 20):

1. *Productos genéricos*. Son sistemas autónomos e independientes que se producen por una organización o empresa de desarrollo y se comercializan en el mercado abierto a cualquier cliente que lo pueda y desee adquirir. Ejemplos de este tipo de productos incluyen aplicaciones para dispositivos móviles, software para computadoras, como bases de datos, procesadores de texto, hojas de cálculos, paquetes de diseño y herramientas de gestión de proyectos. También incluye "aplicaciones" verticales diseñadas para un mercado específico como pueden ser librerías, sistemas de información, sistemas de contabilidad o ERP.
2. *Software personalizado (a medida)*. Éstos son sistemas que se diseñan y desarrollan para un cliente específico. Un contratista de software diseña e implementa el software especialmente para sus clientes; por ejemplo, sistemas de control de dispositivos electrónicos, sistemas específicos escritos para dar soporte de un producto, entre otros.

La diferencia esencial reside en que en los productos genéricos la organización que desarrolla el software controla las especificaciones, lo cual significa que la organización es la que puede planificar su desarrollo; sin embargo, en los productos a medida, las especificaciones se desarrollan y controlan por la organización que adquiere el software y los desarrolladores trabajan con esas especificaciones.

Cristian Pilapanta 1095211-2202112351

18.1.1. Software propietario

El software propietario es de código cerrado, a diferencia del software libre, su uso, distribución y modificación está limitado y restringido a los propietarios de su licencia. El software propietario no se puede copiar ni modificar por personas o entidades ajenas. Está protegido con leyes de copyright y otras leyes de propiedad; generalmente es de pago y su nivel de calidad *suele ser* superior al ofrecido por el software de código abierto. Pese a la creciente implantación del código abierto, el software propietario sigue siendo de uso mayoritario, sobre todo en organizaciones y empresas; tiene una licencia específica por la que el cliente paga una cuota o cantidad fija aceptando ciertas condiciones y limitaciones. Su código fuente no está abierto de modo universal y su uso es restrictivo; algunos ejemplos de este software son:

- *Sistema operativo:* Windows de Microsoft y OS de Mac de Apple.
- *Ofimática*: Office de Microsoft, CRM de Salesforce.com, ERP de SAP, Android de Google, iOs de Apple, etcétera.
- *Navegadores de Internet:* Microsoft Internet Explorer, Mozilla Firefox, Google Chrome, Opera, Safari.
- *Software de antivirus:* Panda, Kaspersky, Norton, Avira, McAffe, Avast.
- *Diseño gráfico:* Corel Draw, Adobe Photoshop CC, Adobe Illustrator, Canva, Pixlr.

18.1.2. Software abierto: código abierto (*open source*)

El desarrollo de software de código abierto (*open source*) es un enfoque en que el código fuente de un sistema de software se publica y se invita a voluntarios a participar en el proceso de desarrollo (Raymond 2001), sus raíces son el software libre de la *Free Software Foundation* (www.fsf.org) –sus orígenes se remontan a los años 80 y a su fundador, Richard Stallman que, a su vez, creó el proyecto GNU y la licencia GPL– que abogaba para que el código fuente no fuese propietario sino que siempre estuviese disponible para que los usuarios lo pudieran examinar y modificar como desearan. En su origen, el software de código abierto nació con unos principios de colaboración, asociación, intercambio de información, acceso rápido, menor costo de la tecnología, etcétera, y como evolución del software libre, con ciertas semejanzas pero también bastantes diferencias.

El término *open source* (código abierto) fue acuñado por Christine Peterson, el 3 de febrero de 1998, en aquel entonces Directora Ejecutiva de Foresight Institute. Su concepto de código abierto, implicaba promoción de software seguro, distribuido y desarrollado libremente, de modo que permitiera:

- Posibilidad de descarga del software y pago por los servicios.
- Base del producto gratuita y módulos de pago.

- Modelo *freemium:* una versión gratuita y otra de pago por funcionalidades avanzadas.
- Producto gratuito y actualizaciones de pago.
- Licencias abiertas para negocios *business source*.

El software abierto es la columna vertebral de Internet y de la ingeniería de software; por lo tanto, encontramos infinidad de programas de código abierto; entre ellos están: Linux, que es el sistema operativo más ampliamente utilizado en servidores, el servidor web de Apache, Java, el IDE de Eclipse, MySQL (en gestión de bases de datos), el sistema operativo Android (instalado en millones de dispositivos móviles), entre otros. La mayoría de los grandes fabricantes de la industria informática como IBM y Oracle soportan el movimiento *open-source* y muchos de sus productos software son de código abierto.

En la actualidad es una tendencia creciente en las empresas y en el caso del software de Big Data, las tecnologías fundamentales de Big Data y de programación avanzada, utilizan código abierto, tales como Hadoop, Spark, Hive, Python, etcétera.

Licencias *open-source*

Existe un gran debate sobre la licencia de software (Sommerville, pp. 220-221). Aunque uno de los principios fundamentales del desarrollo de *open-source* es que el código fuente debe estar disponible libremente, no significa que el desarrollador pueda hacer lo que crea conveniente con ese código. Legalmente, el desarrollador del código (bien una persona o una empresa) es propietario del código. Existe un gran movimiento y tendencias sobre las licencias de software que puedan ser abiertas, propietarias y que se puedan vender como sistemas de fuente cerrados.

La mayoría de las licencias de *open-source* se certifican o distribuyen en tres modelos:

1. La GNU General Public License (GPL). Esta licencia significa que si se utiliza software de fuente abierta está licenciado bajo GPL y por lo tanto ese software debe ser de código abierto.
2. La GNU Lesser General Public License (LGPL). Ésta es una variante de la licencia GPL en donde se pueden escribir componentes que enlazan el código fuente sin tener que publicar la fuente. Sin embargo, si se cambia la componente licenciada, entonces se debe publicar éste como un *open-source.*
3. El Berkeley Standard Distribution (BS) License. Es una licencia no recíproca, que significa que no se está obligado a reeditar cualquier cambio o modificaciones realizadas, al código *open-source*. Se puede incluir el código en sistemas propietarios que son vendidos. Si se utilizan componentes de código abierto se debe agradecer al creador original del código.

La licencia MIT es una variante de la licencia BSD con condiciones similares.

18.2. Proceso de desarrollo de software

Un proceso de desarrollo de software (o *proceso de software*) es un conjunto de actividades relacionadas que conduce a la producción de un sistema de software. El proceso no es único y no es aplicable universalmente; es utilizado por diferentes empresas y depende del tipo de software que se está desarrollando, los requerimientos del cliente y las habilidades de las personas que lo escriben. Aunque existen diversos procesos de software, todos deben incluir de alguna forma las siguientes cuatro actividades fundamentales de ingeniería de software (Sommerville 2016: 44):

1. *Especificaciones del software.* La funcionalidad del software y las restricciones en su funcionamiento deben estar definidos.
2. *Desarrollo del software.* El software debe ser producido para cumplir las especificaciones.
3. *Validación del software.* El software debe ser validado para asegurar que hace lo que el cliente desea.
4. *Evolución del software.* El sistema debe evolucionar para cumplir los cambios que necesite el cliente.

Otras subactividades deben incluir validación de requerimiento, diseño arquitectónico y pruebas (*testing*) de unidad. Y otras actividades como gestión de la configuración del proyecto y planificación de proyectos que soportan actividades de productos. El proceso de software es un conjunto de actividades necesarias para transformar las ideas iniciales del usuario que desea automatizar un determinado trabajo, en software.

Cristian Pilapanta 1095211-2202112351

18.2.1. Actividades del proceso de desarrollo de software

Los procesos de software real son secuencias entremezcladas de técnicas colaborativas y gerenciales con el objetivo global de especificación, diseño, implementación y prueba de un sistema de software. Actualmente, los desarrolladores de software pueden hacer uso de un gran rango de herramientas de software para apoyarse, tales como sistemas de administración de requerimientos, editores de modelos, editores de programas, herramientas automatizadas de pruebas y depuradores.

Las cuatro actividades de proceso, especificación, desarrollo y validación se organizan de modo diferente en los distintos procesos de desarrollo. Así en el modelo en cascada se organizan en secuencias mientras que en el modelo incremental son intercaladas. El modo de ejecución de estas actividades depende del tipo de software que se está desarrollando, las experiencias y competencias de los desarrolladores y el tipo de organización o empresa que desarrolla el software.

Especificación del software

La especificación del software también conocida como ingeniería de requerimientos o requisitos es el proceso de comprensión y definición de cuáles servicios se requieren del sistema, así como la identificación de las restricciones del funcionamiento y desarrollo del mismo.

La ingeniería de requerimientos es una etapa especialmente crítica del proceso de software, ya que los errores cometidos en esta etapa conducen inevitablemente a problemas posteriores en el diseño y en la implementación. El proceso de ingeniería de requerimientos debe dar como resultado un documento consensuando que especifique los requerimientos que se han de cumplir por parte de los socios y partes interesadas del sistema.

Los requerimientos se presentan normalmente en dos niveles de detalle: 1) Los usuarios finales y clientes necesitan una descripción de alto nivel de los requerimientos, 2) Los desarrolladores del sistema necesitan una especificación más detallada. Las tres actividades principales del proceso de ingeniería de requerimientos (Sommerville 2016: 55) son:

1. *Obtención y análisis de requerimientos.* Éste es el proceso de deducir los requerimientos del sistema a través de observaciones de sistemas existentes, discusiones con usuarios y compradores potenciales, analistas de tareas, etc. Esto puede implicar el desarrollo de uno o más modelos de sistema y prototipos.
2. *Especificaciones de requerimientos.* Es la actividad de traducir la información reunida durante el análisis en un documento que define un conjunto de requerimientos que pueden ser; los requerimientos de usuario y los requerimientos del sistema. Los primeros son sentencias abstractas de requerimientos del sistema para el cliente y usuario final; y los segundos son una descripción más detallada de la funcionalidad que se proporcionará.
3. *Validación de requerimientos.* Verifica los requerimientos en función del realismo, consistencia y completitud. Inevitablemente, durante este proceso se descubren los errores en los documentos de requerimientos y por lo tanto se deben modificar para corregir.

Desarrollo del software: diseño e implementación

La etapa de implementación del desarrollo de software es el proceso de desarrollar un sistema ejecutable para entregar al cliente. A veces esta fase implica separar las actividades del diseño del software y la programación. Un diseño de software es la descripción de la estructura del software a ser implementado, los modelos y estructuras de datos utilizados por el sistema, las interfaces entre componentes del sistema y, a veces, los algoritmos utilizados.

Las actividades en el proceso de diseñar varían dependiendo del tipo de sistema a desarrollar. Podemos encontrar cuatro actividades en el proceso de diseño para un sistema de información (Sommerville, 2016: 57).

1. Diseño arquitectónico.
2. Diseño de bases de datos.
3. Diseño de interfaces.
4. Selección y diseño de componentes (se puede utilizar el modelo UML, vea los capítulos 15 a 17).

Cristian Pilapanta 1095211-2202112351

Validación del software

La validación del software implica verificación y validación, está concebida para mostrar un sistema conforme a sus especificaciones y cumple las expectativas de los clientes del sistema. La principal técnica de validación se refiere a las pruebas (*testing*) del programa donde se ejecuta el sistema, utilizando datos simulados. La validación implica también procesos de chequeo, como inspecciones y revisiones en cada etapa del proceso de software desde la definición de los requerimientos al desarrollo de programas, aunque el mayor tiempo del proceso de verificación y validación se gasta en pruebas del programa. Las etapas de prueba del programa son:

1. *Prueba de componentes.* Las personas que desarrollan el sistema prueban los componentes uno a uno de manera independiente. Los componentes pueden ser entidades como funciones o clases de objetos o se pueden agrupar entidades coherentes y relacionadas entre sí.
2. *Prueba del sistema.* Los componentes del sistema se integran para crear un sistema completo. Este proceso busca encontrar errores que procedan de interacciones entre componentes o problemas de interfaces con los componentes.
3. *Prueba del cliente.* Es la etapa final del proceso de pruebas antes de que el sistema sea aceptado para su uso operacional. El sistema se comprueba por el cliente (o clientes potenciales) con datos proporcionado por él mismo y no con datos de test simulados. Las pruebas del cliente pueden revelar problemas de requerimientos si el sistema no cumple las necesidades del usuario y el rendimiento no es aceptable.

Evolución del software

La flexibilidad del software es una de las razones principales por las que cada vez se incorpora más software en sistemas grandes y complejos. Una vez que se ha decidido sobre la fabricación del hardware, es muy costoso hacer cambios en su diseño. Sin embargo, los cambios en el software se pueden realizar en cualquier momento, durante o después del desarrollo del mismo; incluso los cambios más costosos suelen ser más económicos que los correspondientes en el hardware del sistema.

Históricamente siempre ha existido una división entre el proceso de desarrollo del software y el proceso de evolución del sistema (mantenimiento del software). Existen opciones diversas sobre el mantenimiento, pero lo verdaderamente real es que esta etapa requiere grandes conocimientos y habilidades y en ocasiones puede entrañar restos importantes como en el desarrollo original del software.

El desarrollo y mantenimiento del software es un *continuum* y en lugar de considerarlos como dos procesos independientes es mejor considerarlos como un proceso evolutivo dentro de la ingeniería del software.

18.3. Ciclo de vida del desarrollo de software

El ciclo de vida del desarrollo de software (SDLC, software developmente life cycle) es un proceso sistemático para construcción de software que asegura la calidad y exactitud del software construido. El proceso tiene como objetivo principal producir software de alta calidad que cumpla todas las expectativas previstas. Debe ser completo en tiempo y en costes. Consta de un plan detallado que explica cómo planificar, diseñar, construir y mantener un software específico y se organiza en diferentes etapas o fases. El ciclo de vida del software, SDLC, tiene su propio proceso y entregables que han de alimentar a la siguiente fase.

El desarrollo y construcción del software es un proceso continuo que se inicia desde el momento en que se toma la decisión de lanzar el proyecto y termina en el momento de la puesta en marcha del proyecto final en un sistema, organización o empresa. Existen numerosos modelos o metodologías del ciclo de vida del desarrollo de software, aunque todos ellos tienen etapas o fases básicas comunes. Las etapas más comunes de un proceso de desarrollo de software son:

1. Planificación.
2. Análisis de requerimientos.
3. Diseño.
4. Desarrollo (implementación).

Cristian Pilapaña 1095211-2202112351

5. Pruebas (*testing*).
6. Despliegue (instalación).
7. Operaciones (uso) y mantenimiento.

Planificación

Es la fase inicial y en ella se establece el descubrimiento, identificación y definición del ámbito del proyecto con el objetivo de describir el curso de las acciones a seguir. En esta fase se hace un estudio de viabilidad o factibilidad, para determinar la necesidad del nuevo sistema a realizar. Se examinan los aspectos del cliente y se buscan posibles soluciones. Se debe realizar una estimación de los recursos necesarios para la nueva solución y se deben considerar factores como: tiempo completo del proyecto, costes, personal y cualquier otro factor de interés. Es necesario desarrollar un plan del proyecto que se debe presentar a la Dirección y/o Administración o Gerencia para su estudio y valoración. La Gerencia debe verificar los requerimientos de los usuarios finales para asegurar que están dentro de los ámbitos del proyecto. Cuando se obtiene la aprobación del proyecto se pasa a la siguiente fase.

Actividades a realizar:

- Determinación del ámbito del proyecto (delimitación).
- Realización de un estudio de viabilidad.
- Análisis de riesgos asociados con el proyecto.
- Evaluación de costes del proyecto.
- Planificación temporal y definición de las etapas del proyecto.
- Asignación de recursos a las distintas etapas del proyecto.

Análisis de requerimientos

En esta etapa es necesario definir los problemas específicos que tiene el cliente; recolectar y analizar todos los requerimientos del sistema o proyecto, y ser claro para ayudar a comprender las necesidades del cliente y descubrir lo que realmente requiere. Es necesario describir las características que debe tener el proyecto, y comprender de manera adecuada los requerimientos.

El objetivo final de esta etapa es reunir información suficiente para la realización del proyecto. Deben existir suficientes detalles de modo que no se pierda ninguno de ellos. Por lo tanto, es necesario utilizar diferentes técnicas como entrevistas individuales o en grupo, test, encuestas, recolección de información de diferentes fuentes de datos, incluido, la Web y proyectos propios o de la competencia que tengan éxito y sean abiertos; de igual forma se han de utilizar herramientas de modelado del sistema y metodologías de análisis de requerimientos.

Se deben conocer e identificar a las personas afectadas por el proceso, desde el cliente que paga el proyecto hasta usuarios finales y terceras personas o partes interesadas (*stakeholders*), así como organizaciones y empresas, de modo que se garantice que todos los participantes del proyecto han comprendido con claridad las tareas a realizar y cómo se han de implementar todos los requerimientos. Es importante que los usuarios finales participen adecuadamente en esta fase del proyecto.

En esta etapa puede haber la necesidad de especialistas de **aseguramiento de calidad** (*QA, quality assurance*) que a su vez puedan participar en otras fases del proceso.

Diseño

En la fase de diseño es necesario representar las características de los proyectos que permitan su implementación efectiva. Los desarrolladores deben diseñar la *arquitectura del proyecto* teniendo presente todas las diferentes cuestiones técnicas que deben ser examinadas por todas las partes interesadas. Así, se han de definir las tecnologías a utilizar en el proyecto, las cargas de los equipos, limitaciones, marcos de tiempo y presupuestos. Las decisiones más apropiadas del proyecto se deben hacer de acuerdo con los requerimientos definidos.

Los modelos que se utilizan en la fase de diseño representan las características del proyecto que permitirán su implementación efectiva. Un software bien diseñado debería ser modular en lugar de monolítico; sus módulos deben

ser cohesivos (encargados de una tarea concreta y sólo de una) y estar acoplados débilmente entre sí (para facilitar el mantenimiento del software).

Desarrollo (implementación)

En esta fase se crea el producto software. El desarrollo real del proyecto comienza cuando todos los requerimientos se han aprobado y se dispone de un diseño eficiente. Los administradores del proyecto ajustan el entorno de software, los desarrolladores de los sistemas de bases de datos, de infraestructuras de Big Data, aplicaciones de escritorio o móviles, etcétera, los programadores de sistemas y de aplicaciones diseñan las interfaces del usuario del programa y comienzan la escritura del código fuente teniendo presentes los requerimientos definidos anteriormente y el diseño realizado.

En la mayoría de los casos, en esta fase se dividirá el trabajo en unidades y se configurarán los equipos que trabajen en cada una de ellas. Se asignarán a cada unidad, los profesionales individuales y con experiencias adecuadas. La programación en sí, implicará cuatro etapas:

- Desarrollo del algoritmo.
- Escritura del código fuente.
- Compilación o interpretación, según el lenguaje de programación elegido.
- Pruebas y depuración del programa.

Cristian Pilapaña 1095211-2202112352

Pruebas (*testing*)

La fase de pruebas incluye el proceso de depuración. Todo el flujo de código debe fluir durante el desarrollo, se detecta su funcionamiento, se documenta y se pasa de nuevo a los desarrolladores para fijar y establecer la versión final. El proceso de prueba se repite hasta que todos los aspectos críticos se han eliminado y el flujo de trabajo del software es estable.

En esta fase es necesario hacer pruebas de validación de los datos, así como verificar el código y detectar errores semánticos, los más difíciles de detectar y que podrán producir graves problemas si no se consigue su detección. Es necesario verificar las especificaciones del diseño, comprobar los resultados con las pruebas necesarias y comparar los diferentes resultados. En esta fase se disponen de herramientas de pruebas y depuraciones que suelen venir incorporadas en los entornos integrales de desarrollo (EID).

Despliegue (instalación)

Una vez terminado y validado el programa, y comprobado que no existen aspectos críticos en vigor, es el momento de preparar el lanzamiento o despliegue a los usuarios finales, mediante la instalación adecuada del software en su versión final (*release*).

El equipo de desarrollo de software debe realimentar al usuario, atender consultas y dar soporte durante el tiempo de explotación. En esta fase también se incluyen la actualización de los componentes seleccionados para asegurarse que el software está actualizado hasta la fecha de entrega e instalación y es invulnerable a brechas de seguridad.

En esta fase se debe entrenar a los usuarios. La propiedad del proyecto de software se transfiere al cliente y se debe estar atento a la realimentación para tomar las medidas correctoras que fueran necesarias.

Mantenimiento (uso y operaciones)

A medida que el cliente continúa usando el nuevo software, surgirá la necesidad de hacer cambios y mejoras. Con los nuevos requisitos se necesitará una funcionalidad adicional. El ajuste fino del software puede también realizarse con la finalidad de mejorar el rendimiento o desempeño. La realización de actualizaciones programadas puede asegurar la detección de brechas de seguridad y cualquier otra necesidad que surja.

La fase de mantenimiento consume un alto porcentaje de los recursos de desarrollo de software (estadísticas fiables consideran cifras muy altas el coste de la fase de mantenimiento comparado con el coste total del proyecto).

Por esta razón suele considerarse a esta fase como la más importante del ciclo de vida del software hasta su retirada final del mercado o del cliente. Se pueden considerar tres modalidades de mantenimiento:

- *Mantenimiento correctivo*. Elimina los defectos que se detectan durante la vida útil del producto.
- *Mantenimiento adaptativo*. El producto software se adapta a nuevas necesidades y, en particular a actualizaciones del sistema operativo, aplicaciones o entornos de hardware diferentes (nuevos servidores, computadoras personales, dispositivos terminales, etcétera).
- *Mantenimiento perfectivo.* Es un nuevo modelo que añade nuevas funcionalidades al producto software cuando, a su vez, se añaden nuevas características deseables para la mejora del producto ya existente.

Existen dos niveles en la construcción de programas: aquéllos relativos a pequeños programas (los que normalmente realizan programadores individuales) y aquellos que se refieren a sistemas de desarrollo de programas grandes (*proyectos de software*) y que, por lo general, requieren un equipo de programadores en lugar de personas individuales. El primer nivel se denomina *programación a pequeña escala*; el segundo nivel se denomina *programación a gran escala.* La programación en pequeña escala se preocupa de los conceptos que ayudan a crear pequeños programas —aquellos que varían en longitud desde unas pocas líneas a unas pocas páginas—. En estos programas suele requerirse claridad, precisión mental y técnica. En realidad, el interés mayor desde el punto de vista del futuro programador profesional está en los programas de gran escala que requiere de unos principios sólidos y firmes de lo que se conoce como *ingeniería de software* y que constituye un conjunto de técnicas para facilitar el desarrollo de programas de computadora. Estos programas o mejores proyectos de software están realizados por equipos de personas dirigidos por un director de proyectos (analista o ingeniero de software) y los programas pueden tener más de 100.000 líneas de código.

La técnica utilizada por los desarrolladores profesionales de software es comprender lo mejor posible el problema que se está tratando de resolver y crear una solución de software apropiada y durante el **proceso de desarrollo de software**. Un producto de software se desarrolla en varias etapas desde su concepción inicial hasta llegar al producto terminado para su uso regular. Esta secuencia de etapas se conoce como *ciclo de vida del software.*

Los productos de software pueden requerir años de desarrollo y también los esfuerzos de muchas personas, analistas, programadores, usuarios, entre otros. Un producto de software con éxito se utilizará por muchos años después de su primera versión. Durante su periodo de uso, versiones nuevas o actualizadas se lanzarán de modo que irán conteniendo la fijación y corrección de los diferentes errores que hayan surgido. Por esta razón, es importante diseñar y documentar el software de modo que se pueda comprender y mantener fácilmente después de su versión inicial. Esto es especialmente importante, ya que las personas que mantienen el software pueden no haber estado implicadas en el diseño original.

Cristian Pilapanta 1095211-2202112352

18.4. Modelos de procesos de desarrollo de software

Los modelos generales de proceso tradicionales (también llamados paradigmas de proceso) muestran las actividades y sus secuencias, así como los roles de las personas implicadas en las actividades. Existe una gran diversidad de modelos de procesos generales de software que se han de ajustar, en cualquier caso, a las características del producto, el proyecto y la empresa. Los modelos más utilizados son:

- ***En cascada (waterfall).*** Es el modelo tradicional y por excelencia. Tiene las actividades de procesos fundamentales: especificación, desarrollo, evolución y mantenimiento y se representan como etapas de procesos independientes tales como:
  - Especificación de requerimientos.
  - Diseño de software.
  - Implementación.
  - Pruebas (*testing*).
- ***Modelo en V.***
- ***Incremental.*** Este enfoque intercala las actividades de especificación, desarrollo y validación. El sistema se desarrolla como una serie de versiones (incrementos) donde cada versión agrega funcionalidad a la versión anterior. El desarrollo incremental se basa en la idea de desarrollo de una implementación inicial, realimentando de unos

a otros usuarios y evolucionando el software a través de diferentes versiones hasta que se ha desarrollado el sistema requerido. Las actividades de especificación, desarrollo y validación se entremezclan entre sí, realimentándose y no actuando de modo independiente. El modelo incremental, en la actualidad, es un enfoque muy usual en el desarrollo de sistemas de aplicación y productos de software.

- ***Iterativo.*** El método más usual para el desarrollo de software es adoptar un ciclo de vida iterativo, desarrollando un producto de software en etapas o ciclos. Cada etapa es una *miniversión* del modelo en cascada, con énfasis en las diferentes actividades del ciclo de vida. Al final de cada ciclo hay una revisión con los usuarios para obtener realimentación que se tendría en cuenta en la siguiente etapa. Este último modelo se conoce como *desarrollo iterativo e incremental.* Pressman considera diferentes modelos de procesos incrementales: *modelo DRA (desarrollo rápido de aplicaciones), prototipado (construcción de prototipos), modelo en espiral.* Considera también modelos especializados de procesos, basados en componentes, métodos formales y orientados a aspectos.
- ***Espiral.***
- ***Ágiles.***
- ***DevOps.***
- ***Integración y configuración.*** Este enfoque se centra en la disponibilidad de componentes o sistemas reutilizables. El proceso de desarrollo del sistema se centra en la configuración de esos componentes para su uso en una nueva configuración y su integración en un sistema.

Cristian Pilapanta 1095211-2202112352

18.4.1. Métodos ágiles

El modelo en cascada, pese a su amplia adopción, ofrece dificultades para su aplicación en sistemas grandes y complejos. Por esta circunstancia, comenzaron a emerger metodologías que pudieran introducir desarrollos iterativos y ligeros, y terminó en la publicación del *Manifiesto Ágil* (*Agile Manifesto*) que se plasmó en el desarrollo de una serie de principios (doce en origen) que sirvió para promover los modelos ágiles. El Manifiesto del Desarrollo de Software Ágil se redactó por un grupo de desarrolladores de software en 2001, y planteaba el contraste entre el método en cascada, de facto en una época el método de desarrollo estándar y un nuevo método ágil. Este manifiesto se centró en afrontar los problemas clave del modelo "en cascada" y que se manifestaba en la entrega de software. El modelo ágil es una combinación del modelo iterativo e incremental.

El desarrollo de software rápido se conoce como desarrollo o métodos ágiles. Estos métodos ágiles son diseñados para producir software útil rápidamente; son métodos de desarrollo incremental donde los incrementos son pequeños y, normalmente, se crean y hacen disponibles nuevas versiones del sistema a los clientes cada dos o tres semanas. Estos métodos implican a los clientes en el proceso de desarrollo para obtener realimentaciones rápidas cambiando requerimientos. Minimizan la documentación utilizando comunicaciones informales en lugar de reuniones formales con documentos escritos.

Los métodos ágiles entrañan que el software debe ser desarrollado y entregado incrementalmente; se basan en procesos ágiles diferentes, pero que comparten un conjunto de principios basados en el manifiesto ágil, por lo que tienen muchos aspectos comunes. Los métodos ágiles tienen éxito en dos tipos de desarrollo de sistemas (Sommerville 2017: 76):

1. Desarrollo de productos donde una compañía está desarrollando un producto de tamaño pequeño o medio para venta. Todos los productos de software con aplicaciones se desarrollan utilizando un enfoque ágil.
2. Desarrollo de sistemas de clientes en una organización, donde el cliente se implica en el proceso de desarrollo y donde hay pocos grupos de interés y regulaciones que afecten al software.

Los dos métodos ágiles más populares son:

- *EXtreme Programming (XP).* El término y modelo fue acuñado por Kent Beck en 1998 y se apoya en el desarrollo iterativo conducido a niveles "extremos".
- *Método ágil Scrum.* Fue desarrollado en 2001 por Schwaber y Beedle en 2001 y ampliado en 2003 por Rubin siguiendo los principios del método ágil. Se centra alrededor de iteraciones limitadas en el tiempo conocidas como *Sprint.*

El modelo más popular de metodologías ágiles es **Scrum** y, también, el más ampliamente adoptado.

18.4.2. Prototipado

Un prototipo es una versión anterior de un software del sistema que se utiliza para demostrar conceptos, probar opciones de diseño y encontrar más conceptos del problema y su posible solución. El desarrollo iterativo rápido del prototipo es esencial de modo que los costes son controlados y los *stakeholders* del sistema pueden experimentar con el prototipo anterior en el proceso de software.

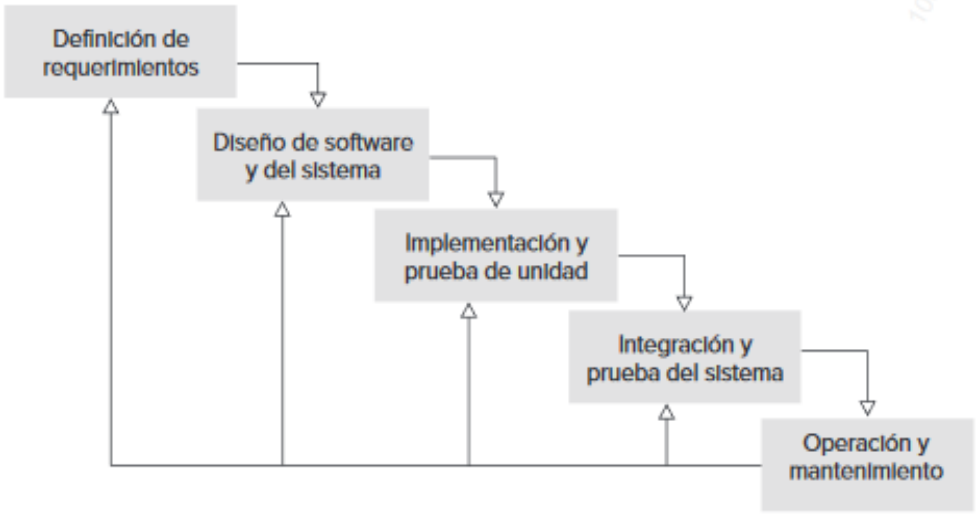
Un prototipo de software se puede utilizar en un proceso de desarrollo de software para ayudar a anticipar los cambios que se puedan requerir (Sommerville, 2017: 62).

18.5. Ciclo de vida del software "en cascada" (*waterfall*)

El ciclo de vida del software en cascada fue el primer modelo publicado de proceso de desarrollo de software derivado de modelos de procesos de ingeniería utilizados en grandes proyectos de ingeniería militar (Royce, 1970). Pese a su antigüedad sigue siendo un modelo muy utilizado y presenta el proceso de desarrollo de software como un número definido de etapas (figura 18-1 Sommerville).

18.5.1. Modelo *waterfall* de Sommerville

Actualmente, el modelo *waterfall de Sommerville* sigue siendo un modelo muy utilizado y muy eficaz en procesos de aprendizaje. Presenta el proceso de desarrollo de software como un número definido de etapas (figura 18.1). Se conoce como ciclo de vida del software y en cascada ya que la salida de una fase es la entrada de la siguiente. En principio, se planifica y prevé todas las actividades del proceso antes de arrancar el desarrollo de software. Las etapas del modelo en cascada reflejan directamente las actividades fundamentales de software, observe la figura 18.1.



Fuente: (Sommerville 2016: 47)

Figura 18.1. Modelo de ingeniería de software *waterfall* (en cascada) de Sommerville.

1. *Análisis y definición de requerimientos.* Los servicios del sistema, restricciones y objetivos se establecen por consultas con usuarios de sistemas, se definen en detalle y sirven como una especificación del sistema.
2. *Diseño de software y sistema*. El proceso de diseño del sistema asigna los requerimientos a los sistemas de hardware o software, establece una arquitectura de sistema global, e implica la identificación y descripción de las abstracciones de software fundamentales y sus relaciones.

3. *Implementación y pruebas de unidad.* Durante esta etapa, el diseño de software es realizado como un conjunto de programas o unidades de programa. Las pruebas de unidad (*testing*) verifican que cada una cumple su especificación.
4. *Integración y pruebas del sistema.* Las unidades de programa individuales o programas se integran y prueban como un sistema completo para asegurar que los requerimientos del software han sido cumplidos. Después de las pruebas, el sistema de software se entrega al cliente.
5. *Operación y mantenimiento.* Normalmente, ésta es la fase del ciclo de vida más largo. El sistema se instala y pone en funcionamiento y uso práctico. El mantenimiento implica corregir errores que no se descubrieron en etapas anteriores del ciclo de vida, mejorando la implementación de las unidades del sistema y mejorando (enriqueciendo) los servicios del sistema a medida que se descubren nuevos requerimientos.

En principio, el resultado de cada fase del modelo "en cascada" consiste en uno o más documentos que son aprobados. La etapa siguiente no debe comenzar sino hasta que se termine la fase anterior. En el desarrollo de hardware, donde están implicados existen altos costes de fabricación tiene mucho sentido; sin embargo, en el desarrollo de software estas etapas se traslapan y se nutren mutuamente de información. Durante el diseño, se identifican los problemas con los requerimientos; durante la codificación del diseño se encuentran problemas de diseño, etcétera. El proceso de software, en la práctica, nunca es un modelo lineal, sino que implica la realimentación de una fase a otra.

A medida que emerge nueva información en una etapa de proceso, los documentos producidos en etapas anteriores se deben modificar para reflejar los cambios requeridos del sistema. Por ejemplo, si se descubre que un requerimiento es demasiado caro de implementar, el documento de requerimientos se debe cambiar para eliminar ese requerimiento. Sin embargo, esto requiere la aprobación del cliente y retrasa el proceso de desarrollo global. Como resultado, clientes y desarrolladores pueden liberar prematuramente la especificación del software de modo que no se hagan cambios futuros. Esta situación puede significar que la resolución de los problemas se deje para más tarde, los problemas se ignoren o se programe a destiempo.

Durante la fase final del ciclo de vida (*operación y mantenimiento*) el software se pone en uso, se descubren los errores y omisiones en los requerimientos de software originales; emergen errores de diseño y programas, y se debe identificar la necesidad de una nueva funcionalidad. Por lo tanto, el sistema debe evolucionar para permanecer útil. La realización de estos cambios (mantenimiento del software) puede implicar la repetición de etapas de proceso anteriores. En realidad, el software debe ser flexible y acomodar sus cambios a medida que se necesiten.

El modelo en cascada no es el modelo de proceso correcto en situaciones donde la comunicación del equipo informal es posible y los requerimientos del software cambian rápidamente. El desarrollo iterativo y los métodos ágiles son mejores para estos sistemas.

Cristian Pilaparta 1095211-2202112352

18.5.2. Modelo de cascada *waterfall* de Pressman

Una variante del modelo tradicional del ciclo de vida "en cascada" la propone Pressman. Considera cinco etapas, con variantes de actividades dentro de cada una y con nombres diferentes de las mismas. En la primera fase de *Comunicación* se inicia el proyecto y se recolectan todos los requerimientos; la salida de esta etapa es la entrada de la siguiente *Planificación*, donde se realiza una estimación, previsiones y seguimiento del proyecto; en la etapa de *Modelado* se lleva a cabo un análisis previo a la realización del diseño que también se desarrolla en esta fase. La propuesta de modelo se introduce a la siguiente fase de *Construcción* donde ya se debe codificar y a continuación realizar las pruebas. Por último, la etapa de *Despliegue*, se entrega ya el producto software al cliente, se da soporte y la realimentación necesaria; es decir la operación y mantenimiento que denominan otros modelos.

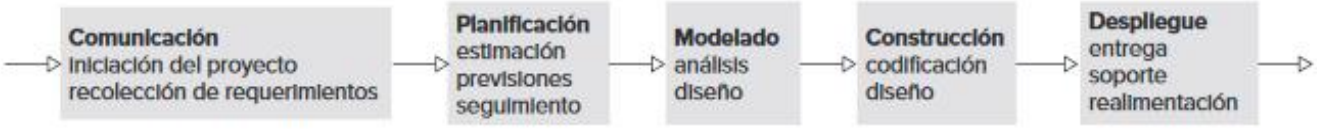


Figura 18.2. Modelo del ciclo de vida "en cascada" de Pressman.

18.5.3. Modelo tradicional de ciclo de vida del software "en cascada"

La versión más simple y utilizada, como ya se ha comentado con anterioridad es el *modelo en cascada*, en el cual se ejecutan las actividades de modo secuencial, de manera que el resultado de cada una de ellas es la entrada de la siguiente. Gráficamente se representa con una flecha que apunta de una etapa a la siguiente.

La figura 18.3 muestra el modelo del *ciclo de vida tradicional del software en cascada que constituye la espina dorsal de la metodología de la programación y* que se compone de seis fases o etapas claves: *Análisis, Diseño, Implementación (construcción), Pruebas, Instalación (despliegue)* y *Mantenimiento.* Las actividades a realizar en cada una de las etapas en el modelo en cascada son:

1. **Análisis de requerimientos.** Esta etapa se divide en Análisis de requisitos y Análisis. En el primero se definen y determinan los requisitos del sistema (análisis y especificaciones). Una vez que éstos se han especificado, comienza la subetapa de análisis, cuyo objetivo es determinar cuidadosamente los requisitos de entrada y salida del sistema y su interacción con el usuario.
2. **Diseño.** Una vez que se conocen los requisitos, el proceso de diseño determina cómo construir un sistema que cumpla estos requisitos. Se define la arquitectura del sistema: componentes, interfaces, relaciones y comportamiento, las funciones y los datos (o clases, en programación orientada a objetos) así como los algoritmos de las funciones.
3. **Implementación (construcción).** El diseño terminado se traduce en el código del programa que implementará el software a desarrollar. La construcción puede utilizar diferentes lenguajes de programación (C, C++, Java, C#, Python, etcétera) y sistemas de gestión de bases de datos, para diferentes partes del sistema. Se codifican las funciones y clases individuales (en el caso de programación orientada a objetos) en el lenguaje de programación elegido.
4. **Pruebas (*testing*).** El sistema se comprueba para asegurar que cumple los requisitos del usuario y que funciona adecuadamente. Las funciones y las clases se prueban aisladas y como unidades.
5. **Instalación (despliegue).** Una vez que el sistema (el producto de software terminado) se ha probado satisfactoriamente, se entrega al cliente y se instala para su utilización posterior.
6. **Mantenimiento.** Ya que se ha entregado el producto se deben detectar posibles fallas y eliminarlas (*mantenimiento correctivo*). Ciertos aspectos del comportamiento del sistema pueden no haberse implementado en su totalidad (por ejemplo, por costes o restricciones de tiempo) y se deben corregir durante la fase de mantenimiento (*mantenimiento correctivo*). El entorno operativo puede cambiar durante su vida útil produciendo cambios en los requisitos que han de adaptarse y acomodarse en su evolución y desarrollo (*mantenimiento adaptativo*).

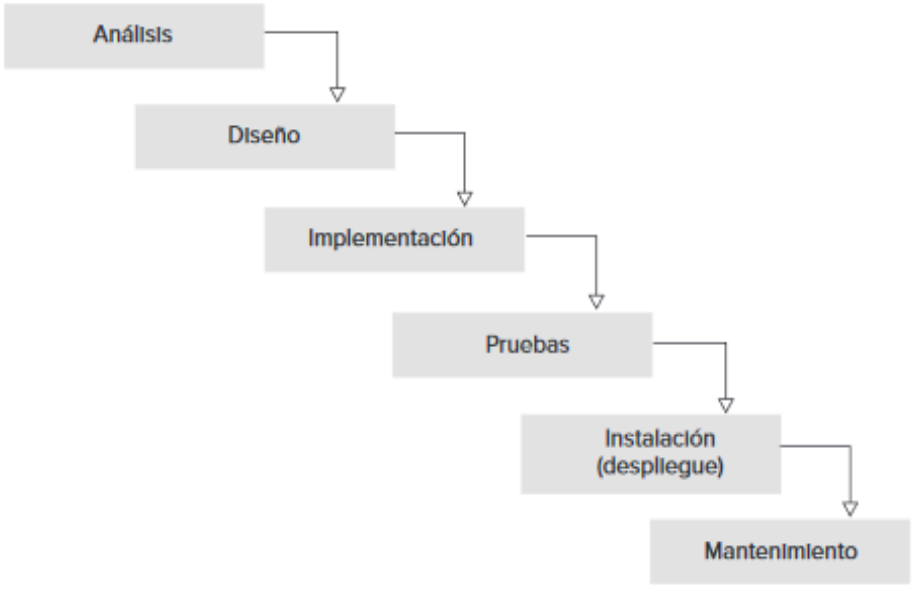


Figura 18.3. Ciclo de vida del software en cascada.

Cristian Pilapanta 1095211-2202112352

El ciclo de vida tradicional (en cascada) se ha utilizado durante muchos años y continúa utilizándose, pero es objeto de muchas críticas ya que presenta algunos problemas, a veces difíciles de resolver, tales como:

- Los proyectos reales rara vez se someten a un ciclo de vida secuencial. Fases del proyecto se solapan y algunas actividades tienen que ser repetidas.
- Las iteraciones son casi inevitables, ya que las insuficiencias o deficiencias en el análisis de requisitos pueden hacerse evidentes durante el diseño, construcción o pruebas.
- Gran parte del tiempo transcurre entre la fase de especificaciones de requisitos y la instalación final. Los requisitos, inevitablemente, cambian en el transcurso del tiempo y eso implica que las operaciones reales se vean afectadas y no sean fáciles las modificaciones necesarias.
- El modelo tradicional no da respuesta fácil a los cambios en los requisitos del cliente o en las tecnologías durante el proyecto. Una vez que se han tomado las decisiones arquitectónicas, son difíciles de cambiar. Una innovación tecnológica suele ser difícil de incorporar ya que casi siempre requerirá rehacer muchos de los trabajos de análisis y diseño.

Por estas razones el ciclo de vida del software es un proceso iterativo, de modo que se modificarán las sucesivas etapas en función de la modificación de las especificaciones de los requisitos producidos en la fase de diseño o implementación, o una vez que el sistema se ha implementado y probado pueden aparecer errores que es necesario corregir y depurar, y que requieren la repetición de etapas anteriores. Por lo tanto es muy común el uso de bucles de realimentación entre etapas, aunque a veces las iteraciones resultantes son muy costosas. Muchas veces, una deficiencia importante en el análisis de requisitos se descubre durante la construcción, de modo que la redefinición de requisitos puede conducir a nuevas implementaciones que no siempre son fáciles de construir. Es por ello que, en ocasiones, suele asignarse a las diferentes etapas, equipos especializados; es decir, el análisis lo hace un equipo especializado, otro el diseño y otro las pruebas.

La metodología de la programación tradicional debe contemplar las etapas clásicas del tradicional ciclo de vida del software y que describiremos en detalle a continuación para dar reglas y consejos de programación en las diferentes etapas ya descritas con anterioridad.

Cristian Pilaparita 1095211-2202112352

18.6. Metodología de la programación: análisis de especificaciones y requerimientos

La primera etapa en la producción de un sistema de software es decidir exactamente *qué* se supone que debe hacer el sistema; esta etapa se conoce también como *análisis de requisitos* o *especificaciones* y por esta circunstancia muchos tratadistas suelen subdividir la etapa en otras dos:

- Análisis y definición del problema (*requisitos*).
- Especificación de requisitos *(especificaciones)*.

La parte más difícil en la tarea de crear un sistema de software es definir cuál es el problema y a continuación especificar lo que se necesita para resolverlo. Normalmente la definición del problema comienza analizando los requisitos del usuario, pero con frecuencia, éstos suelen ser imprecisos y difíciles de describir. Se deben especificar todos los aspectos del problema, pero la mayoría de las veces, las personas que describen el problema no son programadores y eso hace imprecisa la definición. La fase de especificación requiere normalmente la comunicación entre los programadores y los futuros usuarios del sistema e iterar la especificación hasta que tanto el *especificador* como los usuarios estén satisfechos de las especificaciones y hayan resuelto el problema normalmente.

En la etapa de especificaciones puede ser muy útil para mejorar la comunicación entre las diferentes partes implicadas construir un prototipo o modelo sencillo del sistema final; es decir, escribir un programa prototipo que simule el comportamiento de las partes del producto software deseado. Por ejemplo, un programa sencillo —incluso ineficiente— puede demostrar al usuario la interfaz propuesta por el analista. Es mejor descubrir cualquier dificultad o cambiar su idea original ahora que después de que la programación se encuentre en estado avanzado o, incluso, terminada. El modelado de datos es una herramienta muy importante en la etapa de definición del problema. Esta herramienta es muy utilizada en el diseño y construcción de bases de datos.

Tenga presente que el usuario final, normalmente, no conoce con exactitud lo que desea haga el sistema. Por consiguiente, el analista de software o programador, en su caso, debe interactuar con el usuario para encontrar lo que el usuario *deseará* haga el sistema. En esta etapa se debe responder a preguntas tales como:

- ¿Cuáles son los datos de entrada?
- ¿Qué datos son válidos y cuáles no son válidos?
- ¿Quién utilizará el sistema: especialistas cualificados o usuarios cualesquiera (sin formación)?
- ¿Qué interfaces de usuario se utilizarán?
- ¿Cuáles son los mensajes de error y de detección de errores deseables? ¿Cómo debe actuar el sistema cuando el usuario cometa un error en la entrada?
- ¿Qué hipótesis son posibles?
- ¿Existen casos especiales?
- ¿Cuál es el formato de la salida?
- ¿Qué documentación es necesaria?
- ¿Qué mejoras se introducirán —probablemente— al programa en el futuro?
- ¿Qué tan rápido debe ser el sistema?
- ¿Cada cuándo debe cambiarse el sistema después de que se haya entregado?

El resultado final de la fase de análisis es un documento de *especificación de los requisitos del software*. Al contrario que la fase informal de requisitos, el documento de especificaciones describe explícitamente la funcionalidad del producto —es decir, con precisión, lo que se supone que hace el producto— y lista cualquier restricción que deba cumplir. El documento de especificaciones incluye las entradas al producto y las salidas requeridas, éste constituye un contrato. La fase de especificaciones tiene dos salidas principales, la primera es el documento de especificaciones (especificaciones) y la segunda es un plan de gestión del proyecto software.

El documento de especificaciones de requisitos para un nuevo producto de software debe ser generado al principio del proyecto y tanto los usuarios como los diseñadores deben revisar y aprobar dicho documento.

Cristian Pilapanta 1095211-2202112352

EJEMPLO 18.1

Sistema de nóminas de una empresa.

Las entradas deben incluir los rangos o escalas de nómina de cada empleado, los datos de periodos trabajados en la empresa, así como información de los archivos de personal, de modo que se puedan calcular correctamente los impuestos. La salida serán los cheques o transferencias bancarias así como informes de deducciones de cuotas de Seguridad Social. Además, el documento de especificaciones incluye las estipulaciones que debe cumplir el producto para manipular correctamente un amplio rango de deducciones, como pagos de seguros médicos, cuotas a sindicatos o contribuciones a planes de pensiones del empleado.

- El análisis del problema requiere asegurar que el problema está claramente definido y comprendido. Por ello se debe entender cuáles son las salidas requeridas y cuáles son las entradas necesarias.
- Descripción del problema previa y detalladamente.
- Prototipos de programas pueden clarificar el problema.

EJEMPLO 18.2

Definir el documento de especificación de requisitos de un programa que manipule el conjunto de los nombres de los alumnos y números de teléfono de un determinado curso de una facultad de ingeniería, con el objeto de mantenerles informados mediante mensajes de texto SMS de cualquier noticia, calificación, calendario de exámenes, etcétera, relativas al curso académico donde están matriculados.

El programa debe poder insertar nuevas entradas en el directorio o agenda, recuperar o modificar una entrada del directorio, además de la funcionalidad de "enviar el mensaje de texto". Un sistema para encontrar las especificacio-

nes que conduzcan a la elaboración del documento de especificaciones podría ser responder a alguna de estas preguntas u otras similares:

- ¿Existe una lista inicial de nombres y números de teléfonos registrada de modo escrito o bien en un archivo electrónico?
- En caso de no existir un archivo electrónico, los datos escritos en papel se deben introducir todos a la vez y de modo interactivo.
- En el caso de que exista una versión inicial de un archivo electrónico, ¿qué tipo de archivo es? Archivo binario, de texto, etcétera.
- Si el archivo es de texto, ¿cuáles son las convenciones de formato? Los nombres tienen 30 caracteres, el número de teléfono tiene 9, 11..., dígitos (incluye el código del país, de provincia o departamento), es celular (móvil) o fijo.
- ¿Cuál es el formato de los nombres: primer apellido, segundo apellido, nombre, ¿o al contrario nombre, apellido primero y apellido segundo?
- ¿Es posible que un alumno tenga asociado más de un número de teléfono? En caso afirmativo, cuando se introduzcan o recuperen números, ¿cuál número se recupera primero, cuál segundo...?
- Cuando se recupera el nombre de un alumno, ¿se debe visualizar el nombre, el número o números de teléfono, o sólo el número de teléfono...?
- ¿Qué hacer si al introducir un nuevo nombre, éste ya existe o resulta que existe otro alumno con el mismo nombre?
- ¿Qué se debe hacer para modificar el número de teléfono de un alumno, o un nuevo número si se admiten más de uno en la lista?
- ¿...?

Es decir, en un problema, teóricamente sencillo, se deben plantear numerosas cuestiones en la descripción del problema inicial que conduzcan al documento de especificación de requisitos. Muchas de estas preguntas son relativas a detalles de datos de entrada o de salida, errores potenciales, formatos de entrada y de salida, etcétera. Naturalmente, mientras más claro sea el documento de especificación de requisitos, más fácil será analizar y diseñar el problema.

Una vez que los requisitos del sistema se han especificado, comienza la etapa de *análisis*. Antes de iniciar el diseño de una posible solución del problema, se debe estar seguro de que el problema está totalmente comprendido. Si la especificación de requisitos se ha realizado cuidadosamente, el análisis será fácil, si quedan cuestiones sin resolver, deberán solucionarse antes de entrar en la nueva etapa de diseño. En la empresa y en la industria, el analista de sistema junto con los programadores y los usuarios deben considerar si existe un paquete comercial de software que cumpla con los requisitos (es una alternativa al desarrollo del software a medida). Deben determinar el impacto del nuevo producto de software en los sistemas informáticos de la organización y en este caso cuál es el nuevo hardware y software que se necesita para ejecutar el nuevo sistema. En caso de que la solución sea diseñar un nuevo producto de software, se debe determinar la fiabilidad que se debe alcanzar, estimación de los costos y beneficios, así como una previsión de tiempo de desarrollo. También se debe definir cuál es el mejor método de diseño a realizar. Es necesario diseñar e implementar cada programa, pero en el análisis, un objetivo es determinar con fiabilidad los requisitos de entrada/salida del sistema y sus interacciones con el usuario. Debe también pensar en romper el sistema en pequeños componentes que sean fáciles de diseñar y codificar independientemente. Para ello necesitará identificar los módulos o componentes que constituyen el sistema y especificar las interacciones entre ellos. Una vez que se ha terminado este proceso se debe comenzar con la fase de diseño.

18.7. Diseño

La especificación de un sistema indica *lo que* éste debe *hacer*; la etapa de diseño indica *cómo* debe hacerse. Para un sistema pequeño, la etapa de diseño puede ser tan sencilla como escribir un algoritmo en *pseudocódigo*. Para un sistema grande, esta etapa incluye también la fase de diseño de algoritmos, pero incluye el diseño e interacción de un número de algoritmos diferentes, con frecuencia sólo bosquejados, así como una estrategia para cumplir todos los detalles y producir el código correspondiente.

Cristian Pilapanta 1095211-2202112352

Arrancando con las especificaciones, el equipo de diseño determina la estructura interna del producto. Los diseñadores descomponen el producto en *módulos*, piezas independientes de código con interfaces bien definidas al resto del producto. Un objeto es un tipo específico de módulo; la interfaz de cada módulo, es decir, los argumentos que se pasan al módulo y los argumentos que se devuelven del módulo se deben especificar en detalle.

Una vez que el equipo ha completado la descomposición en *módulos (diseño arquitectónico)* se realiza el *diseño detallado*. Para cada módulo se seleccionan los algoritmos y las estructuras de datos elegidas.

Es necesario determinar si se pueden utilizar programas o subprogramas que ya existen o es preciso construirlos totalmente. El proyecto se tiene que dividir en módulos utilizando los principios de diseño descendente. A continuación, se debe indicar la interacción entre módulos; un diagrama de estructuras proporciona un esquema claro de estas relaciones.[2]

En este punto es importante especificar claramente no sólo el propósito de cada módulo, sino también el flujo *de datos* entre ellos. Por ejemplo, se debe responder a las siguientes preguntas: ¿Cuáles datos están disponibles al módulo antes de su ejecución? ¿Qué supone el módulo? ¿Qué hacen los datos después de que se ejecuta el módulo? Por consiguiente, se deben especificar en detalle las hipótesis, entrada y salida para cada módulo. Un medio para realizar estas especificaciones es escribir una precondición, que es una descripción de las condiciones que deben cumplirse al principio del módulo y una *postcondición*, que es una descripción de las condiciones al final de un módulo. Por ejemplo, se puede describir un procedimiento que ordena una lista (un array) de la forma siguiente:

```
procedimiento ordenar (E/S arr: A; E entero: n)
{Ordena una lista en orden ascendente
   precondición: A es un array de n enteros, 1<= n <= Max.
   postcondición: A[1] <= A[2] <...<= A[n], n es inalterable}
```

Por último, se puede utilizar pseudocódigo[3] para especificar los detalles del algoritmo. Es importante que se emplee bastante tiempo en la fase de diseño de sus programas. El resultado final de diseño descendente es una solución fácil de traducir en estructuras de control y estructuras de datos de un lenguaje de programación específico, *por ejemplo*, Java o C.

El diseño es una actividad de *sólo ingeniería*. Su entrada principal es la especificación (o el documento de requisitos) que se traducirá en un documento de diseño, escrito por los programadores o ingenieros de software.

Desde el punto de vista estricto de programación, en esta etapa se construye (diseña) el algoritmo que se utilizará para resolver el problema. La solución normalmente se obtiene por una serie de refinamientos que comienza con el algoritmo inicial encontrado en la fase de análisis hasta que se obtiene un algoritmo completo y aceptable.

El gasto de tiempo en la fase de diseño será ahorro de tiempo cuando escriba y depure su programa.

Una vez que se ha entendido el problema en profundidad y se ha elegido el enfoque global a aplicar, el ingeniero de software (el programador en su defecto) debe plantearse cuál enfoque de diseño aplicar: diseño descendente u orientado a objetos.

En el diseño descendente, un sistema se rompe en un conjunto de subsistemas más pequeños, cada uno de éstos se rompe en componentes más pequeños y así sucesivamente, hasta encontrar unos pequeños y sencillos de codificar fácilmente.

En el diseño orientado a objetos, el desarrollador identifica un conjunto de objetos y especifica su interacción (vea el ejemplo 18.2).

[2] Para obtener más información sobre este tema de diagramas de estructuras, puede consultar estas obras nuestras: *Fundamentos de programación*, 2a. edición, McGraw-Hill, 1992; *Problemas de metodología de la programación*, McGraw-Hill, 1992, o bien la obra *Programación en C*. Joyanes y Zahonero, McGraw-Hill, 2001.

[3] Para consultar el tema del *pseudocódigo*, véanse las obras: *Fundamentos de programación. Algoritmos y estructuras de datos*, 2a. edición, McGraw-Hill, 1996, de Luis Joyanes, y *Fundamentos de programación. Libro de problemas*, McGraw-Hill, 1996, de Luis Joyanes, Luis Rodríguez y Matilde Fernández.

Cristian Pilaparta 1095211-2202112352

EJEMPLO 18.3

Escribir un programa que construya un directorio telefónico de alumnos con sus nombres y números de teléfono de modo que se les puedan enviar mensajes de texto SMS con noticias, calificaciones, fechas de exámenes, anuncios de conferencias, etcétera.

Diseño descendente

El problema de construir un directorio telefónico se descompone a su vez en, por ejemplo, cuatro subproblemas: `Lectura del directorio inicial`, `Introducir una nueva entrada`, `Editar una entrada` y `Recuperar y visualizar una entrada`.

Diseño orientado a objetos

En la definición del problema se detectan los nombres: `Directorio`, `Entrada` y `Archivo`, que constituyen las clases. `Directorio` contiene funciones que leen o escriben información de la clase `Archivo`. `Archivo` y `Directorio` contienen entradas con los campos `Nombre` y `NúmerodeTeléfono`. Por consiguiente, el diagrama de clases para resolver el problema es muy simple.

El actor `Usuario` envía una orden a la clase `Directorio`, con el objeto de activar el programa. El `Directorio` contiene un conjunto de `Entradas`. `Archivo`, a su vez, contiene un conjunto de entradas, y por último el `Directorio` lee datos de un archivo y escribe datos en el mismo.

18.7.1. Diseño de algoritmos

A lo largo de todo el libro se han dado técnicas y directivas para el diseño de algoritmos mediante herramientas como pseudocódigo o diagramas de flujo. En la práctica sería deseable disponer de un método para escribir algoritmos, pero, en la realidad, no existe ningún algoritmo que sirva para realizar dicha escritura. El diseño de algoritmos es un proceso creativo; sin embargo, existen una serie de pautas o líneas a seguir que ayudarán al diseño del algoritmo (vea la tabla 18.1).

Tabla 18.1. Pautas a seguir en el diseño de algoritmos.

1	Formular una solución precisa del problema que debe solucionar el algoritmo.
2	Ver si existe ya algún algoritmo para resolver el problema o bien se puede adaptar uno ya existente (*algoritmos conocidos*).
3	Buscar si existen técnicas estándar que se puedan utilizar para resolver el problema.
4	Elegir una estructura de datos adecuada.
5	Dividir el problema en subproblemas y aplicar el método a cada uno (*diseño descendente*).
6	Si todo lo anterior falla, comience de nuevo en el paso 1.

De cualquier forma, antes de iniciar el diseño del algoritmo es preciso asegurarse de que el programa está bien definido:

- Especificaciones precisas y completas de las entradas necesarias.
- Especificaciones precisas y completas de la salida.
- ¿Cómo debe reaccionar el programa ante datos incorrectos?
- ¿Se emiten mensajes de error?, ¿se detiene el proceso?, etcétera.
- Conocer cuándo y cómo debe terminar un programa.

Cristian Pilaparita 1095211-2202112352

18.8. Implementación (codificación)

La etapa de *implementación* (**codificación**) traduce los algoritmos del diseño en un programa escrito en un lenguaje de programación. Los algoritmos y las estructuras de datos realizadas en pseudocódigo deben traducirse a un lenguaje que entienda la computadora.

La codificación tiene que realizarse en un lenguaje de programación; los lenguajes clásicos más populares son C y los tradicionales Pascal, Fortran, Cobol y C; los lenguajes orientados a objetos más usuales son **C++**, **Java**, **Python**, **Visual Basic.NET**, **Smaltalk** y **C#**, entre otros.

Si un problema se divide en subproblemas, los algoritmos que resuelven cada subproblema (tarea o módulo) deben ser codificados, depurados y probados de manera independiente.

Es relativamente fácil encontrar un error en un procedimiento pequeño. Es casi imposible encontrar todos los errores de un programa grande que se codificó y comprobó como una sola unidad en lugar de como una colección de módulos (procedimientos) bien definidos.

Las reglas del sangrado (*indentación*) y buenos comentarios facilitan la escritura del código. El *pseudocódigo* es una herramienta excelente que facilita notablemente la codificación.

Codificar la solución consiste en escribir el programa e implementar la solución y se manifiesta en la traducción del algoritmo en un programa comprensible por la computadora.

18.9. Pruebas e integración

La etapa de *pruebas* requiere, como su nombre sugiere, la prueba o verificación del programa de computadora terminado con el objetivo de asegurar lo que hace; de hecho, proporciona una solución al problema. Cualquier error que se encuentre durante esta prueba o test se debe corregir. Cuando los diferentes componentes de un programa se han implementado y comprobado individualmente, el sistema completo se ensambla e integra.

La etapa de pruebas es útil para mostrar que un programa es correcto. Las pruebas nunca son fáciles. Edgar Dirjkstra ha escrito que mientras que las pruebas realmente muestran la *presencia* de errores, nunca pueden mostrar su *ausencia*. Una prueba con "éxito" en la ejecución significa sólo que no se han descubierto errores en esas circunstancias específicas, pero no se dice nada de otras circunstancias. En teoría el único modo que una prueba puede mostrar que un programa es correcto si *todos* los casos posibles se han intentado y comprobado (es lo que se conoce como *prueba exhaustiva*); es una situación técnicamente imposible incluso para los programas más sencillos. Supongamos, por ejemplo, que se ha escrito un programa que calcule la nota media de un examen. Una prueba exhaustiva requerirá todas las combinaciones posibles de notas y tamaños de clases; puede llevar muchos años completar la prueba.

La fase de pruebas es una parte esencial de un proyecto de programación, ya que durante esta fase se deben eliminar tantos errores lógicos como sea posible. En primer lugar, se debe probar el programa con datos de entrada válidos que conducen a una solución conocida. Si ciertos datos deben estar dentro de un rango, es necesario incluir los valores en los extremos finales del rango. Por ejemplo, si el valor de entrada de n cae en el rango de 1 a 10, se debe de asegurar incluir casos de prueba en los que n esté entre 1 y 10. También se deben incluir datos no válidos para comprobar la capacidad de detección de errores. Se deben probar algunos datos aleatorios y por último intentar algunos datos reales. Cuando los diferentes componentes de un programa se han implementado y comprobado individualmente, el sistema completo se ensambla y se integra.

18.9.1. Verificación

La etapa de pruebas debe comenzar tan pronto como sea posible en la fase de diseño y continuar a lo largo de la implementación del sistema. Incluso aunque las pruebas son herramientas extremadamente válidas para proporcionar la evidencia de que un programa es correcto y cumple sus especificaciones, es difícil conocer si las pruebas realizadas son suficientes. Por ejemplo, ¿cómo se puede conocer que son suficientes los diferentes conjuntos de datos de prueba o que se han ejecutado todos los caminos posibles a través del programa?

Cristian Pilaparta 1095211-2202112352

Por esas razones se ha desarrollado un segundo método para demostrar la corrección o exactitud de un programa. Este método, denominado *verificación formal*, implica la construcción de pruebas matemáticas que ayudan a determinar si los programas hacen lo que se supone debe hacer. La verificación formal implica la aplicación de reglas formales para mostrar que un programa cumple su especificación: la verificación. La verificación formal funciona bien en programas pequeños, pero es compleja cuando se utiliza en programas grandes. La teoría de la verificación requiere conocimientos matemáticos avanzados y por otra parte se sale fuera de los objetivos de este libro; por esta razón sólo hemos constatado la importancia de esta etapa.

La prueba de que un algoritmo es correcto es como probar un teorema matemático. Por ejemplo, probar que un módulo es exacto (correcto) comienza con las precondiciones (axiomas e hipótesis en matemáticas) y muestra que las etapas del algoritmo conducen a las postcondiciones. La verificación intenta probar con medios matemáticos que los algoritmos son correctos.

Si se descubre un error durante el proceso de verificación, se debe corregir su algoritmo y posiblemente se deben modificar las especificaciones del problema. Un método es utilizar *invariantes* (una condición que siempre es verdadera en un punto específico de un algoritmo) lo que probablemente hará que su algoritmo contenga pocos errores *antes* de que comience la codificación. Como resultado, se gastará menos tiempo en la depuración de su programa.

18.9.2. Técnicas de pruebas

Una vez que se ha generado el código fuente de un programa es necesario probar el software para detectar o descubrir y corregir la mayor cantidad de errores posibles antes de entregarlo al cliente. El software debe verificar, su exactitud (corrección) mediante técnicas de pruebas *(testing)*. Desgraciadamente las pruebas son una ciencia inexacta; no se puede declarar que una pieza o módulo de software es correcta vía pruebas a menos que éstas sean exhaustivas y en todos los escenarios posibles; pero, incluso en los programas sencillos hay, con frecuencia, cientos de miles de caminos que pueden recorrerse. Por consiguiente, probar todos los posibles caminos a través de un programa complejo es una tarea imposible (Brookshear 05).

Por otra parte, las pruebas representan un reto para los ingenieros de software que han desarrollado metodologías de prueba. Las técnicas de prueba del software proporcionan directrices para pruebas de diseño: 1) comprobar la lógica interna y las interfaces de todo componente del software; 2) comprobar los dominios de entrada y salida del programa para descubrir errores en su función, comportamiento y desempeño.

Ya en 1979, Glen Myers establecía una serie de reglas sobre las pruebas a realizar al software y que aún hoy siguen vigentes (Pressman 05:124):

- Las pruebas consisten en un proceso en el que se ejecuta un programa con la intención de encontrar un error que no se ha descubierto.
- Un buen caso de prueba es aquel en el que hay una gran probabilidad de encontrar un error que aún no se ha descubierto.
- Una prueba con éxito es aquella que encuentra un error que no se descubría.

La observación ha demostrado que los errores en software tienden a ser repetidos. Esto es, la experiencia demuestra que un número pequeño de módulos dentro de un sistema de software grande tiende a ser más problemático que el resto. Por consiguiente, la identificación de estos módulos y la prueba de ellos de modo más exhaustivo puede ayudar a encontrar más errores del sistema que si se probaran todos los módulos de modo uniforme. Esta característica se conoce como *principio de Pareto* (en honor del economista italiano Wilfredo Pareto, 1848-1923, que estudió casos de teorías de población en Italia con muestras similares relativas a Ciencias de la Salud) y demuestra que 80% de los errores descubiertos durante las pruebas en profundidad serán similares o rastreables en 20% de todos los programas. El problema, naturalmente, es aislar estos componentes sospechosos y después probarlos.

Davis propone un conjunto de pruebas (Davis 95) además del ya citado principio de Pareto:

1. Todas las pruebas deben ser rastreables (seguidas) hasta los requisitos del cliente.
2. Se deben planear mucho antes de que comience el proceso de prueba.
3. Deben comenzar "en lo pequeño" y progresar hacia "lo grande".
4. Las pruebas exhaustivas no son posibles.

Cristian Pilapanta 1095211-2202112352

Facilidad de las pruebas de software

El cliente prueba el programa cada vez que lo ejecuta; por consiguiente, se debe ejecutar el programa antes de que llegue al cliente, y el objetivo concreto será encontrar y eliminar todos los errores. La localización de la mayor cantidad de errores requiere técnicas sistemáticas de pruebas. El objetivo de estas técnicas es encontrar errores y la calidad de estas técnicas se mide por su posibilidad de encontrar un error. En consecuencia, se trata de que las pruebas sean fáciles con el objetivo de encontrar la mayor cantidad de errores con un mínimo esfuerzo. Existen numerosas metodologías de prueba de software. Las más utilizadas en las pruebas de módulos o componentes: *ruta básica, caja negra y caja blanca.*

1. *Prueba de la ruta básica.* Consiste en desarrollar un conjunto de datos de prueba que asegure que cada instrucción o sentencia, del software, se ejecute al menos una vez. Las técnicas de teoría de grafos permiten identificar tales conjuntos de datos. Por consiguiente, aunque puede ser imposible asegurar que cada camino a través de un sistema de software sea comprobado, es posible asegurar que cada sentencia dentro del sistema se ejecute al menos una vez durante el proceso de pruebas.
2. *Pruebas de caja negra.* Se aplican a la interfaz del software. No se conocen los detalles internos, se conocen las funciones específicas para las que se diseñó el producto (módulo de software), por lo cual examina su aspecto funcional. En esencia, las pruebas de caja negra se ejecutan desde el punto de vista del usuario y se centran, no en el modo en que el software ejecuta su tarea, sino simplemente en que el software se ejecute correctamente en términos de exactitud y temporalidad *(timeliness)*.
3. *Pruebas de caja blanca.* En este caso se conocen los detalles internos de la implementación y, por consiguiente, los diferentes caminos de ejecución. Las pruebas se basan en un examen próximo al detalle procedimental; se prueban las rutas lógicas del software y la colaboración entre componentes, al proporcionar casos de prueba que utilicen conjuntos específicos de condiciones, bucles, o ambos.

En aplicaciones convencionales de software, éste se prueba desde dos perspectivas diferentes (Pressman 05:419):

1. La lógica interna del programa se prueba mediante técnicas de diseño de casos de prueba de "caja blanca";
2. Los requisitos de software se comprueban empleando técnicas de diseño de casos de prueba de "caja negra".

En el caso de aplicaciones orientadas a objetos, la prueba empieza antes de la existencia del código fuente, pero una vez generado éste, se diseña una serie de pruebas para comprobar operaciones de una clase, relaciones con otras clases, etcétera.

Niveles de las pruebas

Las pruebas del software se pueden realizar desde diferentes niveles teniendo presente el equipo humano que lo ha desarrollado.

- *Pruebas de módulo o clase.* Se realizan de modo independiente, y la lleva a cabo el equipo que ha implementado el módulo. Preferentemente son pruebas de caja blanca.
- *Pruebas de integración.* Son realizadas por los técnicos que realizaron las fases de diseño e implementación y son fundamentalmente pruebas de caja negra. Tratan de probar la interacción entre módulos.
- *Pruebas de aplicación.* Se aplican a los casos de construcción de software complejo y por esta razón suelen realizarlas aquellos técnicos que desarrollaron la fase de análisis.
- *Pruebas beta.* Esta metodología cae dentro de la categoría de caja negra y es utilizada por los desarrolladores de software de ambiente (entorno) PC. Además, proporciona, a una audiencia especializada, una versión preliminar del programa denominada versión beta. El objetivo último es aprender cómo se ejecuta el software en situaciones de la vida real antes de lanzar al mercado la versión final del producto. Las realimentaciones de los usuarios —normalmente personas expertas— además, de la posible detección de errores buscan que los desarrolladores de software comiencen a diseñar productos compatibles con el producto en prueba. Por ejemplo, en el caso de un nuevo sistema operativo, la distribución de una versión beta abierta al desarrollo de software de utilidad (utilidades o utilerías) compatible, de modo que el sistema operativo final aparece rodeado de aplicaciones actuales basadas en ese sistema operativo. Además, claro, las versiones beta buscan impactar en el mercado con acciones específicas de mercadotecnia que influyen considerablemente en las ventas.

Cristian Pilaquinga 1095211-2202112352

18.9.3. Revisión general de validación y verificación de programas

Aunque muchos programadores utilizan indistintamente los términos *prueba* o *comprobación* (*testing*) y *depuración*, son diferentes. La **comprobación (pruebas)** se refiere a las acciones que determinan si un programa funciona correctamente. La **depuración** es la actividad posterior de encontrar y eliminar los errores (*bugs*) de un programa. Las pruebas de ejecución de programas —por lo general— muestran claramente que el programa contiene errores, aunque el proceso de depuración puede, en ocasiones, resultar difícil de seguir y comprender.

No obstante, el análisis anterior no significa que la comprobación sea imposible; al contrario, existen diferentes metodologías formales para las comprobaciones de programas. Una filosofía adecuada para pruebas de programas incluye las siguientes consideraciones:

1. Suponer que su programa tiene errores hasta que sus pruebas muestren lo contrario.
2. Ningún test simple de ejecución puede probar que un programa está libre de error.
3. Trate de someter al programa a pruebas duras. Un programa bien diseñado manipula entradas "con elegancia". Por este término se entiende que el programa no produce errores en tiempo de ejecución ni produce resultados incorrectos; por el contrario, el programa, en la mayoría de los casos, visualizará un mensaje de error claro y solicita de nuevo los datos de entrada.
4. Comenzar la comprobación antes de terminar la codificación.
5. Cambiar sólo una cosa cada vez.

La **prueba de un programa** ocurre cuando se ejecuta y se observa su comportamiento.

Cada vez que se ejecuta un programa con algunas entradas se prueba para ver cómo funciona el trabajo para esa entrada particular. Cada prueba ayuda a establecer que el programa cumple las especificaciones dadas.

Selección de datos de prueba

Cada prueba debe ayudar a establecer que el programa cumple las especificaciones dadas. Parte de la ciencia de *ingeniería de software* es la construcción sistemática de un conjunto de entradas de prueba que es idóneo para descubrir errores.

Para que un conjunto de datos pueda ser considerado como buenos datos de prueba, sus entradas para prueba necesitan cumplir dos propiedades.

Propiedades de buenos datos de prueba

1. Se debe conocer cuál es la salida correcta que debe producir un programa para cada entrada de prueba.
2. Las entradas de prueba deben incluir aquellas entradas que probablemente originen más errores

Se deben buscar numerosos métodos para encontrar datos de prueba que pudieran producir errores. El primer método se basa en identificar y probar entradas denominadas *valores externos*, que son especialmente idóneos para causar errores. **Un valor externo** o **límite** de un problema en una entrada produce un tipo diferente de comportamiento. Por ejemplo, suponiendo que se tiene una función `ver_hora` que tiene un parámetro *hora* y una precondición:

Precondición: Horas está comprendido en el rango 0-23.

Los dos valores límites de `ver_hora` son `hora` igual a 0 (dado que un valor menor de 0 es ilegal) y `hora` igual a 23 (dado que un valor superior a 23-24, ... es ilegal). Puede ocurrir que la función se comporte de modo diferente para horario matutino (0 a 11) o nocturno (12 a 23), entonces 11 y 12 serán valores extremos. Si se espera un comportamiento diferente para `hora` igual a 0, entonces 1 es un valor extremo. En general no existe una definición precisa de valor extremo, pero debe ser aquel que muestre un comportamiento límite en el sistema.

Cristian Pilapanta 1095211-2202112352

Valores de prueba extremos

Si no se pueden probar todas las entradas posibles, se deben probar al menos los valores extremos. Por ejemplo, si el rango de entradas legales va de cero a un millón, asegúrese probar la entrada 0 y la entrada 1000000. Es buena idea considerar también 0.1 y –1 como valores límites siempre que sean entradas legales.

Otra técnica de prueba de datos es la denominada **perfilador** que básicamente considera dos reglas:

1. Asegúrese de que cada línea de su código se ejecuta al menos una vez para algunos de sus datos de prueba. Por ejemplo, puede ser una porción de su código que maneje alguna situación rara.
2. Si existe alguna parte de su código que a veces se salte totalmente, asegúrese, en ese caso, que existe al menos una entrada de prueba que salte realmente esta parte de su código. Por ejemplo, un bucle en el que el cuerpo se ejecute, en ocasiones, cero veces. Asegúrese de que hay una entrada de prueba que produce que el cuerpo del bucle se ejecute cero veces.

Errores de sintaxis (de compilación)

Un *error de sintaxis* o *en tiempo de compilación* se produce cuando existen errores en la sintaxis del programa, tales como signos de puntuación incorrectos, palabras mal escritas, ausencia de separadores (signos de puntuación), o de palabras reservadas. Si una sentencia tiene un error de sintaxis, no puede ser traducida y su programa no se ejecutará.

Los errores de sintaxis son detectados por el compilador.

```
44 Field identifier expected
```

Por lo general, los mensajes de error son fáciles de encontrar. El siguiente ejemplo (en Object Pascal) presenta dos errores de sintaxis: el punto y coma que falta al final de la primera línea y la palabra `WritaLn` mal escrita debería ser `WriteLn`.

```
Suma :- 0
for I:- 0 to 10 do
   Suma:- Suma + A[I];
WritaLn (Suma/10);
```

Errores en tiempo de ejecución

Los errores en tiempo de ejecución —o simplemente de ejecución— (*runtime error*) suceden cuando el programa trata de hacer algo imposible o ilógico; y sólo se detectan en la ejecución. Errores típicos son: la división por cero, intentar utilizar un subíndice fuera de los límites definidos en un array, etcétera.

```
X ← 1/N produce un error si N = 0
```

Los mensajes de error típicos son del tipo:

```
Run-Time error nnn at xxxx:yyyy
```

`nnn` *número de error en ejecución.*
`xxxx:yyyy` *dirección del error en ejecución (segmento y desplazamiento).*

Los errores de ejecución se dividen en cuatro categorías:

- *Errores DOS (números de mensaje).*
- *Errores I/O.*
- *Errores críticos.*
- *Errores fatales.*

Cristian Pilapaña 1095211-2202112352

Errores lógicos

Los errores lógicos son errores del algoritmo o de la lógica del programa; son difíciles de encontrar porque el compilador no produce ningún mensaje de error y se producen cuando el programa es perfectamente válido y produce una respuesta.

Calcular la media de todos los números leídos del teclado

```
Suma ← 0
desde i ← 0 hasta 10 hacer
   Leer (Num)
   Suma ← Suma + Num
fin_desde
Media ← Suma / 10
```

La media está calculada mal, ya que existen once números (0 a 10) y no diez como se ha escrito. Si se desea escribir la sentencia:

```
Salario ← Horas * Tasa
```

y se escribe:

```
Salario ← Horas + Tasa
```

Es un error lógico (+ por *) ya que *a priori* el programa funciona bien y sería difícil a no ser que el resultado fuese obvio, detectar el error.

18.10. Depuración de programas

Una de las primeras cosas que se descubren al escribir programas es que raramente funcionan de manera correcta la primera vez. La ley de Murphy "si algo puede ser incorrecto, lo será" parece estar escrita pensando en la programación de computadoras.

Aunque un programa funcione sin mensajes de error y produzca resultados, puede ser incorrecto. Un programa es correcto sólo *si se producen resultados correctos para todas las entradas válidas posibles*. El proceso de eliminar errores —*bugs*— se denomina **depuración** (*debugging*) de un programa.

Cuando el compilador detecta algo, la computadora visualiza *un mensaje de error*, que indica que se ha producido un error y cuál puede ser la causa posible. Por desgracia, los mensajes de error son, con frecuencia, difíciles de interpretar y, a veces, son engañosos. Los errores de programación se pueden dividir en tres clases: **de compilación** (sintaxis), **en tiempo de ejecución** y **lógicos**.

18.10.1. Localización y reparación de errores

Aunque se sigan todas las técnicas de diseño dadas a lo largo del libro y en este capítulo, en particular, y cualquier otras que haya obtenido por cualquier otro medio (otros libros, experiencias, cursos, etcétera) es prácticamente imposible e inevitable que su programa carezca de errores. Afortunadamente los programas modulares, claros y bien documentados, son ciertamente más fáciles de depurar que aquellos que no lo son. Es recomendable utilizar técnicas de seguridad contra fallos, que protejan contra ciertos errores e informen de ellos cuando se encuentran.

Con frecuencia el programador, pero sobre todo el estudiante de programación, está convencido de la bondad de sus líneas de programa, sin pensar en las múltiples opciones que pueden producir los errores: el estado incorrecto de una variable lógica, la entrada de una cláusula `then` o `else`, la salida imprevista de un bucle por un mal diseño de su contador, etcétera. El enfoque adecuado debe ser seguir la traza de la ejecución del programa utilizando las facilidades de depuración del EID (Entorno Integrado de Desarrollo) o añadir sentencias de escritura que muestren cuál fue la cláusula ejecutada. En el caso de condiciones lógicas, si la condición es falsa cuando se espera que sea verdadera —como el mensaje de error puede indicar— entonces el siguiente paso es determinar cómo se ha convertido en falsa.

Cristian Pilaparia 1095211-2202112352

¿Cómo se puede encontrar el punto de un programa en que algo se ha convertido en una cosa distinta a lo que se había previsto? Se puede hacer el seguimiento de la ejecución o bien paso a paso a través de las sentencias del programa o estableciendo puntos de ruptura (*breakpoint*). Se puede examinar también el contenido de una variable específica ya sea estableciendo inspecciones/observaciones (*watches*) o insertando sentencias `escribir` temporales. La clave para una buena depuración es sencillamente utilizar estas herramientas que indiquen lo que está haciendo el programa.

La idea principal es localizar sistemáticamente los puntos del programa que causan el problema. La lógica de un programa implica que ciertas condiciones sean verdaderas en puntos diferentes del programa (recuerde que estas condiciones se llaman *invariantes*). Un error (*bug*) significa que una condición que pensaba iba a ser verdadera no lo es. Para corregir el error se debe encontrar la primera posición del programa en la que una de estas condiciones difiera de sus expectativas. La inserción apropiada de puntos de ruptura, y de observación o inspección o sentencias `escribir` en posiciones estratégicas de un programa —tal como entradas y salidas de bucles, estructuras selectivas y subprogramas— sirven para aislar sistemáticamente el error.

Las herramientas de diagnóstico deben informar si las cosas son correctas o equivocadas antes o después de un punto dado del programa. Por consiguiente, después de ejecutar el programa con un conjunto inicial de diagnósticos se ha de poder seguir el error entre dos puntos. Por ejemplo, si el programa ha funcionado bien hasta la llamada al procedimiento o función `P1`, pero algo falla cuando se llama al procedimiento `P2`, nos permite centrar el problema entre estos dos puntos, la llamada a `P2` y el punto concreto donde se ha producido el error en `P2`. Este método es muy parecido al de *aproximaciones sucesivas*, es decir, ir acotando la causa posible de error hasta limitarla a unas pocas sentencias.

Naturalmente la habilidad para situar los puntos de ruptura, de observación o sentencias `escribir` dependerá del dominio que se tenga del programa y de la experiencia del programador. No obstante, le damos a continuación algunas reglas prácticas que le faciliten su tarea de depuración.

Cristian Pilapaña 1095211-2202112352

Uso de sentencias escribir

Las sentencias `escribir` pueden ser muy adecuadas en numerosas ocasiones, y sirven para informar sobre valores de variables importantes y la posición en el programa en que las variables toman esos valores. Es conveniente utilizar un comentario para etiquetar la posición.

```
{Posición una}
escribir('Está situado en posición uno del procedimiento Test')
escribir('A=', a, 'B = ', b, 'C = ', c)
```

18.10.2. Depuración de sentencias si-entonces-sino

Es importante situar una parte de ruptura antes de una sentencia `si-entonces-sino` y examinar los valores de las expresiones lógicas y de sus variables. Se pueden utilizar o bien puntos de ruptura o sentencias `escribir` para determinar qué alternativa de la sentencia **si** se toma:

```
//Examinar valores de <condición> y variables antes de si
si <condición>    entonces
   escribir('Condición verdadera: siga camino');
...
si_no
   escribir('Condición falsa: siga camino');
fin_si
```

Depuración de bucles

Situar los puntos de ruptura al principio y al final del bucle y examinar los valores de las variables importantes.

```
//examinar valores de m y n antes de entrar al bucle
desde i = m hasta n hacer
   //Examinar los valores de i y variables importantes
fin_desde
//Examinar los valores de m y n después de salir del bucle
```

Depuración de subprogramas

Las dos posiciones clave para situar los puntos de ruptura son al principio y al final de un subprograma. Se deben examinar los valores de los parámetros en estas dos posiciones utilizando o bien sentencias de escritura o ventanas de inspección u observación (*watches*).

Lecturas de estructuras de datos completos

Las variables cuyos valores son *arrays* u otras estructuras puede ser interesante examinarlas. Para ello se recurre a escribir *rutinas específicas de volcado* (presentación en pantalla o papel) que ejecuten la tarea. Una vez diseñada la rutina se llama a ella desde puntos diferentes según interesa a la secuencia de flujo de control del programa y los datos que sean necesarios en cada caso.

18.10.3. Los equipos de programación

En la actualidad es difícil y raro que un gran proyecto de software sea *implementado* (realizado) por un solo programador; normalmente se asigna a un equipo de programadores, que por anticipado deben coordinar toda la organización global del proyecto.

Cada miembro del equipo es responsable de un conjunto de procedimientos, algunos de los cuales pueden ser utilizados por otros miembros del equipo. Cada uno de estos miembros deberá proporcionar a los otros las especificaciones de cada procedimiento, condiciones *pretest* o *postest* y su lista de parámetros formales; es decir, la información que un potencial usuario del procedimiento necesita conocer para poder ser llamado.

Normalmente, un miembro del equipo actúa como bibliotecario, de modo que a medida que un nuevo procedimiento se termina y comprueba, su versión actualizada sustituye la versión actualmente existente en la librería. Una de las tareas del bibliotecario es controlar la fecha en que cada nueva versión de un procedimiento se ha incorporado a la librería, así como asegurarse de que todos los programadores utilizan la última versión de cualquier procedimiento.

Es misión del equipo de programadores crear librerías de procedimientos, que posteriormente puedan ser utilizadas en otras aplicaciones. Una condición importante que deben cumplir los procedimientos son: *estar comprobados y ahorro de tiempo/memoria*.

Cristian Pilapanta 1095211-2202112352

18.10.4. El depurador

Los Entornos Integrados de Desarrollo (**EID**) tienen un *programa depurador* disponible para ayudarle a depurar un programa; además, le permite ejecutar su programa, una sentencia cada vez, de modo que se pueda ver el efecto de la misma. El depurador imprime un diagnóstico cuando ocurre un error de ejecución, indica la sentencia que produce el error y permite visualizar los valores de variables seleccionadas en el momento del error. Asimismo se puede seguir la pista de los valores de variables seleccionadas durante la ejecución del programa (*traza*), de modo que se pueda observar cómo cambian mientras el programa se ejecuta. Por último, se puede pedir al depurador que detenga la ejecución en determinados puntos (*breakpoints*); en esos momentos se pueden inspeccionar los valores de las variables seleccionadas a fin de determinar si son correctas.

El depurador tiene la gran ventaja de posibilitar la observación de los diferentes valores que van tomando las variables dentro del programa.

18.11. Mantenimiento

Cuando el producto software (el programa) se ha terminado, se distribuye entre los posibles usuarios, se instala en las computadoras y se utiliza (***producción***). Sin embargo, y aunque, *a priori*, el programa funcione correctamente, el software debe ser mantenido y actualizado. De hecho, el costo típico del mantenimiento excede, con creces, el costo de producción del sistema original.

Un sistema de software producirá errores que serán detectados, casi con seguridad, por los usuarios del sistema y que no se descubrieron durante la fase de prueba. La corrección de estos errores es parte del mantenimiento del

software. Otro aspecto de la fase de mantenimiento es la mejora del software añadiendo más características o modificando partes existentes que se adapten mejor a los usuarios.

Otras causas que obligarán a revisar el sistema de software en la etapa de mantenimiento son:

1) Cuando un nuevo *hardware* se introduce, el sistema puede ser modificado para ejecutarlo en un nuevo entorno; 2) Si cambian las necesidades del usuario, suele ser menos caro y más rápido modificar el sistema existente que producir un sistema totalmente nuevo. La mayor parte del tiempo de los programadores de un sistema se gasta en el mantenimiento de los sistemas existentes y no en el diseño de sistemas totalmente nuevos. Por esta causa, entre otras, se ha de tratar siempre de diseñar programas de modo que sean fáciles de comprender y entender (legibles) y fáciles de cambiar.

18.11.1. La obsolescencia: programas obsoletos

La última etapa en el ciclo de vida del software es la evolución del mismo, pasando por su vida útil hasta su *obsolescencia* o fase en la que el software se queda anticuado y es preciso actualizarlo o escribir un nuevo programa sustitutorio.

La decisión de dar de baja un software por obsoleto no es fácil. Un sistema grande representa una inversión enorme de capital que parece, a primera vista, más barato modificar el sistema existente en vez de construir un sistema totalmente nuevo. Este criterio suele ser, normalmente, correcto y por esta causa los sistemas grandes se diseñan para ser modificados. Un sistema puede ser productivamente revisado muchas veces. Sin embargo, incluso los programas grandes se quedan obsoletos por caducidad de tiempo al pasar una fecha límite determinada. A menos que un programa grande esté bien escrito y adecuado a la tarea a realizar, como en el caso de programas pequeños, suele ser más eficiente escribir uno nuevo que corregir el antiguo.

18.11.2. Iteración y evolución del software

Las etapas de vida del software suelen formar parte de un ciclo o bucle, como su nombre sugiere y no son simplemente una lista lineal. Es probable, por ejemplo, que durante la fase de mantenimiento tenga que volver a las especificaciones del problema para verificarlas o modificarlas.

Observe en la figura 18.4 que las diferentes etapas rodean al núcleo documentación. La documentación no es una etapa independiente como se puede esperar sino que está integrada en todas las etapas del ciclo de vida del software.

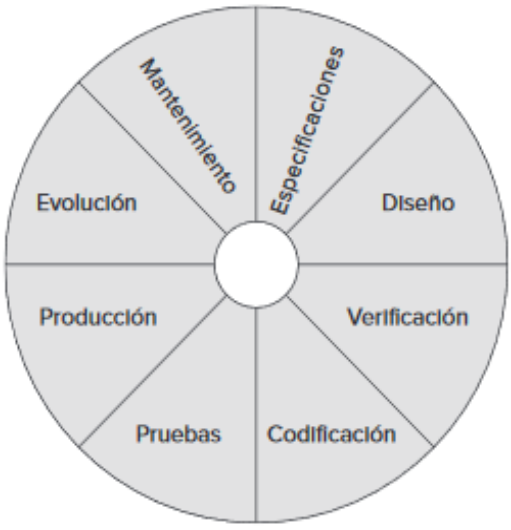


Figura 18.4. Etapas del ciclo de vida del software con la documentación como núcleo aglutinador.

Cristian Pilaparta 1095211-2202112353

18.12. Documentación

Un programa (un paquete de software) de computadora necesita siempre de una documentación que permita a sus usuarios aprender a utilizarlo y mantenerlo. La documentación es una parte importante de cualquier paquete de software y, a su vez, su desarrollo es una pieza clave en la ingeniería de software.

La documentación de un paquete de software se produce, normalmente, para dos fines. Uno, es explicar las características del software y describir cómo utilizarlo, y el otro propósito de la documentación es describir la composición interna del software, de modo que el sistema pueda ser mantenido a lo largo de su ciclo de vida. La primera se denomina documentación del usuario y la segunda, documentación del sistema.

La *documentación del usuario* está diseñada para ser leída por el usuario del software, y, por consiguiente, tiende a ser no técnica. Hoy día, la documentación del usuario se reconoce como una herramienta de marketing. Una buena documentación del usuario, combinada con una interfaz de usuario bien diseñada hace un paquete de software accesible y, por consiguiente, aumenta sus ventas. Por esta razón muchos desarrolladores de software contratan a escritores técnicos para producir esta parte del producto o bien proporcionan versiones preliminares de sus productos a autores independientes de modo que puedan escribir manuales del usuario que estén disponibles en las librerías cuando se lanza el producto al público.

La documentación de usuario, tradicionalmente, toma el formato de libro en papel, aunque cada día es más frecuente incluir el manual en el menú Ayuda del propio programa o como libro electrónico en un CD, DVD o *pendrive*.

La *documentación del sistema* es inherentemente más técnica que la del usuario. Un componente importante de la documentación del sistema es la versión fuente de todos los programas del sistema. Es muy importante que estos programas sean presentados en formato muy legible; por esta razón se requiere un buen uso de la sintaxis y de la gramática del lenguaje de programación de alto nivel, el uso de sentencias comentario adecuadas y en los puntos notables del código, un diseño modular que permita a cada módulo ser presentado como una unidad coherente. Asimismo se requiere seguir convenios de notación, sangrados en las líneas de programa, establecer diferencias en la escritura de nombres de variables, constantes, objetos, clases, etcétera, y convenios que aseguren que todos los programas están documentados suficientemente.

Existen tres grupos de personas que necesitan conocer la documentación del programa: programadores, operadores y usuarios. Los requisitos necesarios para cada uno de ellos suelen ser diferentes, en función de las misiones de cada grupo:

programadores	*manual de mantenimiento del programa.*
operadores	*manual del operador.* operador: persona encargada de correr el programa, introducir datos y extraer resultados.
usuario	*manual del usuario.* usuario: persona o sección de una organización que explota el programa, conociendo su función, las entradas requeridas, el proceso a ejecutar y la salida que produce.

En entornos interactivos, las misiones del usuario y operador suelen ser las mismas. Así pues, la documentación del programa se puede concretar a: Manual del usuario y Manual de mantenimiento.

18.12.1. Manual del usuario

La documentación de un paquete (programa) de software suele producirse con dos propósitos: "uno, es explicar las funciones del software y describir el modo de utilizarlas (*documentación del usuario*) porque está diseñada para ser leída por el usuario del programa; dos, describir el software en sí para poder mantener el sistema en una etapa posterior de su ciclo de vida (*documentación del sistema o de mantenimiento*)".

La documentación de usuario es un instrumento comercial importante. Una buena documentación de usuario hará al programa más accesible y asequible. La documentación del sistema o manual de mantenimiento es por naturaleza más técnica que la del usuario. Antiguamente esta documentación consistía en los programas fuente finales y algunas explicaciones sobre la construcción de los mismos. Hoy día esto ya no es suficiente y es necesario estructurar y ampliar esta documentación.

La documentación del sistema abarca todo el ciclo de vida del desarrollo del software, incluidas las especificaciones originales del sistema y aquéllas con las que se verificó el sistema, los *diagramas de flujo de datos* (**DFD**), *dia-*

Cristian Pilapanta 1095211-2202112353

gramas entidad-relación (**DER**), diccionario de datos y diagramas o cartas de estructura que representan la estructura modular del sistema.

El problema más grave que se plantea es la construcción práctica real de la documentación y su continua actualización. Durante el ciclo de vida del software cambian continuamente las especificaciones, los diagramas de flujo y de **E/R** (*Entidad/Relación*) o el diagrama de estructura; esto hace que la documentación inicial se quede obsoleta o incorrecta y por esta causa la documentación requiere una actualización continua de modo que la documentación final sea lo más exacta posible y se ajuste a la estructura final del programa.

El manual de usuario debe cubrir al menos los siguientes puntos:

- Órdenes necesarias para cargar el programa en memoria desde el almacenamiento secundario (disco) o almacenamiento en la nube (*cloud*) y arrancar su funcionamiento.
- Nombres de los archivos externos a los que accede el programa.
- Formato de todos los mensajes de error o informes.
- Opciones en el funcionamiento del programa.
- Descripción detallada de la función realizada por el programa.
- Descripción detallada, preferiblemente con ejemplos, de cualquier salida producida por el programa.

18.12.2. Manual de mantenimiento (documentación para programadores)

El manual de mantenimiento es la documentación requerida para mantener un programa durante su ciclo de vida. Se divide en dos categorías:

Documentación interna

Esta documentación cubre los aspectos del programa relativos a la sintaxis del lenguaje y está contenida en los *comentarios*, encerrados entre llaves { } o bien paréntesis/asteriscos (* *), o en una línea precedido por //. Algunos tópicos para considerar son:

- Cabecera de programa (nombre del programador, fecha de la versión actual, breve descripción del programa).
- Nombres significativos para describir identificadores.
- Comentarios relativos a la función del programa, así como de los módulos que componen el programa.
- Claridad de estilo y formato (una sentencia por línea, *indentación* [sangrado]), líneas en blanco para separar módulos (procedimientos, funciones, unidades, etcétera).
- Comentarios significativos.

EJEMPLOS

```
var
   real: Radio //entrada, radio de un círculo
...
//Calcular Área
Área ← Pi * Radio * Radio
```

Documentación externa

Documentación ajena al programa fuente, que se suele incluir en un manual que acompaña al programa. La documentación externa debe incluir:

- Listado actual del programa fuente, mapas de memoria, referencias cruzadas, etcétera.
- Especificación del programa: documento que define el propósito y modo de funcionamiento del programa.
- Diagrama de estructura que representa la organización jerárquica de los módulos que comprende el programa.
- Explicaciones de fórmulas complejas.

Cristian Pilapanta 1095211-2202112353

- Especificación de los datos a procesar: archivos externos incluyendo el formato de las estructuras de los registros, campos, etcétera.
- Formatos de pantallas utilizados para interactuar con los usuarios.
- Cualquier indicación especial que pueda servir a los programadores que deben mantener el programa.

18.12.3. Reglas de documentación

Un programa *bien documentado* es aquel que otras personas pueden leer, usar y modificar. Existen muchos estilos aceptables de documentación y, con frecuencia, los temas a incluir dependerán del programa específico. No obstante, señalamos a continuación algunas características esenciales comunes a cualquier documentación de un programa:

1. Un comentario de cabecera para el programa que incluye:
   a) Descripción del programa: propósito.
   b) Autor y fecha.
   c) Descripción de la entrada y salida del programa.
   d) Descripción de cómo utilizar el programa.
   e) Hipótesis sobre tipos de datos esperados.
   f) Breve descripción de los algoritmos globales y estructuras de datos.
   g) Descripción de las variables importantes.
2. Comentarios breves en cada módulo similares a la cabecera del programa y que contenga información adecuada de ese módulo, incluyendo en su caso *precondiciones* y *postcondiciones*. Describir las entradas y cómo las salidas se relacionan con las entradas.
3. Escribir comentarios inteligentes en el cuerpo de cada módulo que expliquen partes importantes y confusas del programa.
4. Describir claramente y con precisión los modelos de datos fundamentales y las estructuras de datos seleccionadas para representarlas, así como las operaciones realizadas para cada procedimiento.

Aunque existe la tendencia entre los programadores y sobre todo entre los principiantes a documentar los programas como última etapa, esto no es buena práctica, lo idóneo es hacerlo a medida que se desarrolla. La tarea de escribir un programa grande se puede extender por periodos de semanas o incluso meses. Esto le ha de llevar a la consideración de que lo que resulta evidente ahora puede no serlo de aquí a dos meses; por esta causa, documentar a medida que se progresa en el programa es una regla de oro para una programación eficaz.

Regla

Asegúrese de que siempre se corresponden los comentarios y el código. Si se hace un cambio importante en el código, asegúrese de que se realiza un cambio similar en el comentario.

18.13. Depuración

Una de las primeras cosas que se descubren al escribir programas es que un programa raramente funciona de manera correcta la primera vez. La ley de Murphy "si algo puede ser incorrecto, lo será" parece estar escrita pensando en la programación de computadoras.

Aunque un programa funcione sin mensajes de error y produzca resultados, puede ser incorrecto. Un programa es correcto sólo *si se producen resultados correctos para todas las entradas válidas posibles*. El proceso de eliminar errores —*bugs*— se denomina **depuración** (*debugging*).

Cuando el compilador detecta algo mal, la computadora visualiza *un mensaje de error*, que indica que se ha producido un error y cuál puede ser la causa posible. Desgraciadamente, los mensajes de error son, con frecuencia, difíciles de interpretar y a veces son engañosos. Los errores de programación se pueden dividir en tres clases: **errores de compilación** (sintaxis), **errores en tiempo de ejecución** y **errores lógicos**.

Cristian Pilapanta 1095211-2202112353

18.13.1. Localización y reparación de errores

Aunque se sigan todas las técnicas de diseño dadas a lo largo del libro y en este capítulo, en particular, y cualquier otras que haya obtenido por cualquier otro medio (otros libros, experiencias, cursos, etcétera) es prácticamente imposible e inevitable que su programa carezca de errores. Afortunadamente los programas modulares, claros y bien documentados, son ciertamente más fáciles de depurar que aquellos que no lo son. Es recomendable utilizar técnicas de seguridad contra fallos, que protejan contra ciertos errores e informen de ellos cuando se encuentran.

Con frecuencia el programador, pero sobre todo el estudiante de programación, está convencido de la bondad de sus líneas de programa, sin pensar en las múltiples opciones que pueden producir los errores: el estado incorrecto de una variable lógica, la entrada de una cláusula `then` o `else`, la salida imprevista de un bucle por un mal diseño de su contador, etcétera. El enfoque adecuado debe ser seguir la traza de la ejecución del programa utilizando las facilidades de depuración del EID (Entorno Integrado de Desarrollo) o añadir sentencias de escritura que muestren cuál fue la cláusula ejecutada. En el caso de condiciones lógicas, si la condición es falsa cuando se espera que sea verdadera —como el mensaje de error puede indicar— entonces el siguiente paso es determinar cómo se ha convertido en falsa.

¿Cómo se puede encontrar el punto de un programa en que algo se ha convertido en una cosa distinta a lo que se había previsto? Se puede hacer el seguimiento de la ejecución paso a paso a través de las sentencias del programa o bien estableciendo puntos de ruptura (*breakpoint*). Se puede examinar también el contenido de una variable específica estableciendo inspecciones/observaciones (*watches*) o bien insertando sentencias `escribir` temporales. La clave para una buena depuración es sencillamente utilizar estas herramientas que indiquen lo que está haciendo el programa.

La idea principal es localizar sistemáticamente los puntos que causan el problema. La lógica de un programa implica que ciertas condiciones sean verdaderas en puntos diferentes del programa (recuerde que estas condiciones se llaman *invariantes*). Un error (*bug*) significa que una condición que pensaba iba a ser verdadera no lo es. Para corregir el error se debe encontrar la primera posición del programa en la que una de estas condiciones difiera de sus expectativas. La inserción apropiada de puntos de ruptura, y de observación o inspección o sentencias `escribir` en posiciones estratégicas de un programa —tal como entradas y salidas de bucles, estructuras selectivas y subprogramas— sirven para aislar sistemáticamente el error.

Las herramientas de diagnóstico han de informarles si las cosas son correctas o equivocadas antes o después de un punto dado del programa. Por consiguiente, después de ejecutar el programa con un conjunto inicial de diagnósticos se ha de poder seguir el error entre dos puntos. Por ejemplo, si el programa ha funcionado bien hasta la llamada al procedimiento o función `P1`, pero algo falla cuando se llama al procedimiento `P2`, nos permite centrar el problema entre estos dos puntos, la llamada a `P2` y el punto concreto donde se ha producido el error en `P2`. Este método es muy parecido al de *aproximaciones sucesivas*, es decir, ir acotando la causa posible de error hasta limitarla a unas pocas sentencias.

Naturalmente la habilidad para situar los puntos de ruptura, de observación o sentencias `escribir` dependerá del dominio que se tenga del programa y de la experiencia del programador. No obstante, le damos a continuación algunas reglas prácticas que le faciliten su tarea de depuración.

Uso de sentencias escribir

Las sentencias `escribir` pueden ser muy adecuadas en numerosas ocasiones. Tales sentencias sirven para informar sobre valores de variables importantes y la posición en el programa en que las variables toman esos valores. Es conveniente utilizar un comentario para etiquetar la posición.

```
{Posición uno}
escribir('Está situado en posición uno del procedimiento Test')
escribir('A=', a, 'B = ', b, 'C = ', c)
```

18.13.2. Depuración de sentencias si-entonces-si no

Situar una parte de ruptura antes de una sentencia `si-entonces-si no` y examinar los valores de las expresiones lógicas y de sus variables. Se pueden utilizar o bien puntos de ruptura o sentencias `escribir` para determinar qué alternativa de la sentencia **si** se toma:

Cristian Pilapanta 1095211-2202112353

```
//Examinar valores de <condición> y variables antes de si
si <condición>    entonces
   escribir('Condición verdadera: siga camino');
   ...
   si_no
      escribir('Condición falsa: siga camino');
fin_si
```

Depuración de bucles

Situar los puntos de ruptura al principio y al final del bucle y examinar los valores de las variables importantes.

```
//examinar valores de m y n antes de entrar al bucle
desde i = m hasta n hacer
   //Examinar los valores de i y variables importantes
fin_desde
//Examinar los valores de m y n después de salir del bucle
```

Depuración de subprogramas

Las dos posiciones clave para situar los puntos de ruptura son al principio y al final de un subprograma. Se deben examinar los valores de los parámetros en estas dos posiciones utilizando o bien sentencias de escritura o ventanas de inspección u observación (*watches*).

Lecturas de estructuras de datos completos

Resulta interesante examinar las variables cuyos valores son arrays u otras estructuras; para ello se recurre a escribir rutinas específicas de volcado (presentación en pantalla o papel) que ejecuten la tarea. Una vez diseñada la rutina se llama a ella desde puntos diferentes según interesa a la secuencia de flujo de control del programa y los datos que sean necesarios en cada caso.

18.13.3. Los equipos de programación

En la actualidad es difícil y raro que un gran proyecto de software sea *implementado* (realizado) por un solo programador. Por lo general, un proyecto grande se asigna a un equipo de programadores, que por anticipado deben coordinar toda la organización global del proyecto.

Cada miembro del equipo es responsable de un conjunto de procedimientos, algunos de los cuales pueden ser utilizados por otros miembros del equipo. Cada uno de estos miembros deberá proporcionar a los otros las especificaciones de cada procedimiento, condiciones *pretest* o *postest* y su lista de parámetros formales; es decir, la información que un potencial usuario del procedimiento necesita conocer para poder ser llamado.

Normalmente, un miembro del equipo actúa como bibliotecario, de modo que a medida que un nuevo procedimiento se termina y comprueba, su versión actualizada sustituye la versión actualmente existente en la librería. Una de las tareas del bibliotecario es controlar la fecha en que cada nueva versión de un procedimiento se ha incorporado a la librería, así como asegurarse de que todos los programadores utilizan la versión última de cualquier procedimiento.

Es misión del equipo de programadores crear librerías de procedimientos, que posteriormente puedan ser utilizadas en otras aplicaciones. Una condición importante que deben cumplir los procedimientos: *estar comprobados y ahorro de tiempo/memoria*.

18.14. Diseño de algoritmos

Tras la fase de análisis, para poder solucionar problemas sobre una computadora debe conocerse cómo diseñar algoritmos. En la práctica sería deseable disponer de un método para escribir algoritmos, pero, en la realidad, no

Cristian Pilaparita 1095211-2202112353

existe ningún algoritmo que sirva para realizar dicha escritura. El diseño de algoritmos es un proceso creativo; sin embargo, existen una serie de pautas o líneas a seguir que ayudarán al diseño del algoritmo (tabla 18.2).

Tabla 18.2. Pautas a seguir en el diseño de algoritmos.

1	Formular una solución precisa del problema que debe solucionar el algoritmo.
2	Ver ya si existe algún algoritmo para resolver el problema o si se puede adaptar uno ya existente (*algoritmos conocidos*).
3	Buscar si existen técnicas estándar que se puedan utilizar para resolver el problema.
4	Elegir una estructura de datos adecuada.
5	Dividir el problema en subproblemas y aplicar el método a cada uno (*diseño descendente*).
6	Si todo lo anterior falla, comience de nuevo en el paso 1.

De cualquier forma, antes de iniciar el diseño del algoritmo es preciso asegurarse de que el programa está bien definido:

- Especificaciones precisas y completas de las entradas necesarias.
- Especificaciones precisas y completas de la salida.
- ¿Cómo debe reaccionar el programa ante datos incorrectos?
- ¿Se emiten mensajes de error? ¿Se detiene el proceso?, etcétera.
- Conocer cuándo y cómo debe terminar un programa.

Cristian Pilaparia 1095211-2202112353

1095211220211235

18.15. Eficiencia

La *eficiencia* es una medida de cantidad de recursos consumidos por el programa. Tradicionalmente, los recursos considerados han sido el tiempo de ejecución y/o el almacenamiento (ocupación del programa en memoria). Mientras menos tiempo se utilice y menor almacenamiento, el programa será más eficiente.

El tiempo y almacenamiento (memoria) de la computadora suelen ser costosos y por ello su ahorro siempre será importante. En algunos casos la eficiencia es críticamente importante: control de una unidad de vigilancia intensiva de un hospital —un retardo de fracciones de segundo puede ser vital en la vida de un enfermo—, un programa de control de roturas en una prensa hidráulica —la no detección a tiempo podría producir grandes inundaciones—, etc. Por el contrario, existirán otros casos en los que el tiempo no será factor importante: control de reservas de pasajeros en una agencia de viajes.

La mejora del tiempo de ejecución y el ahorro en memoria suelen conseguirse con la mejora de los algoritmos y sus programas respectivos. En ocasiones, un simple cambio en un programa puede aumentar la velocidad de ejecución considerablemente. Es muy importante que los programas sean eficientes y legibles.

18.15.1. Eficiencia *versus* legibilidad (claridad)

Las grandes velocidades de los microprocesadores (unidades centrales de proceso) actuales, junto con el aumento considerable de las memorias centrales (cifras típicas usuales superan normalmente los 4 a 16 GB), hacen que los recursos típicos tiempo y almacenamiento no sean, hoy día, parámetros fundamentales para la medida de la eficiencia de un programa.

Por otra parte, es preciso tener en cuenta que —a veces— los cambios para mejorar un programa pueden hacerlo más difícil de comprender: poco legibles o claros. En programas grandes la legibilidad suele ser más importante que el ahorro en tiempo y en almacenamiento en memoria. Como norma general, cuando la elección en un programa se debe hacer entre claridad y eficiencia, generalmente se elegirá la claridad o la legibilidad.

18.16. Portabilidad

Un programa es *transportable* o *portable* si se puede trasladar a otra computadora sin cambios o con pocos cambios apreciables. La forma de hacer un programa transportable es elegir como lenguaje de programación la versión estándar del mismo, en el caso de C y de C++ los estándares reconocidos también por ANSI, es decir, ANSI C++. En el caso de Java y de C#, no existe más problema que la elección del proveedor, dado que ambos lenguajes están sometidos a un proceso de estandarización y de hecho siguen las normas que establecen sus fabricantes principales Oracle y Microsoft.

Conceptos clave

- Abstracción de datos
- Abstracción procedimental
- Ciclo de vida del software
- Clase
- Compatibilidad
- Corrección
- Diseño
- Documentación
- Eficiencia
- Extensibilidad
- Integridad
- Módulos
- Ocultación
- Prueba
- Reutilización
- Robustez
- Transportabilidad
- Verificabilidad

Cristian Pizarro 1085211-202112353

Resumen

El desarrollo de un buen sistema de software se realiza durante el *ciclo de vida*, que es el periodo que se extiende desde la concepción inicial del sistema hasta su eventual retirada de la comercialización o uso del mismo. Las actividades humanas relacionadas con el ciclo de vida implican procesos como análisis de requisitos, diseño, implementación, codificación, pruebas, verificación, documentación, mantenimiento y evolución del sistema y obsolescencia.

Diferentes herramientas ayudan al programador y al ingeniero de software a diseñar una solución para un problema dado. Algunas de estas herramientas son diseño descendente, abstracción procedimental, abstracción de datos, ocultación de la información y programación orientada a objetos.

Una *abstracción procedimental* separa el propósito de un subprograma de su implementación. De modo similar, la *abstracción de datos* se centra en las operaciones que se ejecutan sobre los datos en lugar de centrarse en la manera en que se implementarán las operaciones.

Durante la fase de diseño, la abstracción es uno de los medios más importantes con los que se intenta hacer frente a la complejidad. La innovación de la programación orientada a objetos no reside en la idea de escribir programas utilizando una serie de abstracciones, sino en el uso de clases para gestionar dichas abstracciones.

Los lenguajes modernos facilitan la implementación de abstracciones de datos mediante clases, que pueden definirse como colecciones estructuradas de implementaciones de tipos abstractos cuya potencia reside en la herencia.

Las técnicas orientadas a objetos aumentan la productividad y fiabilidad del desarrollador y facilitan la reutilización y extensibilidad del código.

La construcción de software requiere el cumplimiento de numerosas características. Entre ellas destacan las siguientes:

- *Eficiencia*. La eficiencia de un software es su capacidad para hacer un buen uso de los recursos que manipula.

- *Verificabilidad*. Se refiere a la facilidad de verificación de un software, es decir, es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.
- *Fácil de utilizar.* Un software es fácil de utilizar si se puede comunicar consigo de manera cómoda.
- *Robustez*. Capacidad de los productos software de funcionar incluso en situaciones anormales.
- Existen dos principios fundamentales para conseguir esto: diseño simple y descentralización.
- *Compatibilidad*. Facilidad de los productos para ser combinados con otros.
- *Transportabilidad* (*portabilidad*). Es la facilidad con la que un software puede ser transportado sobre diferentes sistemas físicos o lógicos.
- *Integridad*. Es la capacidad de un software para proteger sus propios componentes contra los procesos que no tenga el derecho de acceder.
- *Corrección*. Capacidad de los productos software de realizar exactamente las tareas definidas por su especificación.
- *Extensibilidad*. Facilidad que tienen los productos de adaptarse a cambios en su especificación.
- *Reutilización*. Capacidad de los productos de ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.

Cristian Pilapanta 1095211-2202112353

Bibliografía y recursos de programación

Cristian Pliparita 1095211-2202112353

ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORAS

Charte, F. y García, L. (2019). *El pasado de la computación personal.* (2a. ed.). Jaén: Uja.

Gil de Lamadrid, J. (2018). *Computer organization. Basic processor structure.* CRC Press.

Null, L. y Lobur, J. (2015). *The essentials of computer organization and architecture.* (4a. ed.). Jones & Bartlett Learning.

Null, L. y Lobur, J. (2018). *The essentials of computer organization and architecture.* (5a. ed.). Jones & Bartlett Learning.

Patterson, D. A. y Hennessy, J. L. (2016). *Computer organization and design. The hardware/software interface.* Morgan Kaufman.

Patterson, D. A. y Hennessy, J. L. (2017). *Computer organization and design: The hardware software interface (RISC-V Edition) solution.* Morgan Kaufman.

Prieto, A. (2006). *Introducción a la informática.* (4a. ed.). McGraw-Hill.

Stallings, W. (2016). *Computer organization and architecture. Designing for performance.* Pearson

Stallings, W. (2016). *Foundations of modern networking. SDN, NFV, QoE, IoT and cloud.* Pearson.

Stallings, W. (2018). *Operating systems: internals and design principles.* Pearson.

Stallings, W., Brown, L. (2018). *Computer security: principles and practice.* Pearson.

INGENIERÍA DE SOFTWARE

Allbee, B. (2018). *Hands-On software engineering with python.* Borminghan: Packt Publishing.

Osetskyi, V. (2017). *SDLC models explained: agile, waterfall, V-shaped, iterative, spiral.* https://medium.com/existek/sdlc-models-explained-agile-waterfall-v-shaped-iterative-spiral-e3f012f390c5.

Pressman, S. R. (2014). *Software engineering: a practitioner's.* Approach. McGraw-Hill.

Sommerville, I. (2016). *Software engineering.* (10a. ed.). Pearson.

Unhelkar, B. (2018). *Software engineering with UML. CRC Press.*

FUNDAMENTOS Y METODOLOGÍA DE LA PROGRAMACIÓN

Gries, P., Campbell, J., Montojo, J. *Practical programming. An introduction to computer science using Python 3.6.* (3a. ed.). Practical Programmers versión 1.0, diciembre 2017.

Joyanes, L. (2010). *Fundamentos generales de programación.* CDMX (México): McGraw-Hill.

Joyanes, L., Rodríguez, L. y Fernández, M. (2000). *Fundamentos de programación. Libro de problemas.* (2a. ed.). McGraw-Hill.

Knuth, D. E. (1997). *The art of computer programming,* vol. 2. Addison-Wesley.

Knuth, D. E. (1998). *The art of computer programming,* vol. 3. Addison-Wesley.

Lizcano, D. y Alonso, F. (2018). *Fundamentos de la programación.* CEF/UDIMA.

Zelle, J. M. (2017). *Python programming: an introduction to computer science.* (3a. ed.). Franklin, Beedle (www.fbeedle.com).

ALGORITMOS Y ESTRUCTURAS DE DATOS

Esel, F., Rohaut, S. *Algoritmia: técnicas fundamentales en programación. Ejemplos en Python.* Ediciones ENI, 2019.

Wirth, N. (1999). *Algoritmos + Estructuras de Datos = Programas.* Ediciones del Castillo.

LENGUAJES DE PROGRAMACIÓN C/C++/C#

Deitel, P. y Deitel, H. (2015). *C. How to program.* (8a. ed.). Pearson.
Deitel, P. y Deitel, H. (2016). *C# 6 for programmers.* (6a. ed.).
Deitel, P. y Deitel, H. (2017). *C++. How to program.* (11a. ed.). Pearson.
Joyanes, L. (2014). *Programación en C/C++, Java y UML.* (2a. ed.). México: McGraw-Hill.
Joyanes, L. y Zahonero, I. (2014). *Programación en C, C++, Java y UML.* (2a. ed.). CDMX (México): McGraw-Hill.
Kernighan, B. W. y Ritchie, D. M. (1999). *El lenguaje de programación C.* (2a. ed.). Prentice-Hall Iberoamericana.
Savitch, W. (2006). *Problem solving with C++.* (6a. ed.). Pearson.
Schildt, H. (2014). C++. *A beginner´s guide.* (3a. ed.). McGraw-Hill.
Stroustrup, B. (1994). *The design and evolution of C++.* Addison-Wesley.
Stroustrup, B. (2000). *Programming. Principles and practice using C++.* Addison-Wesley.
Stroustrup, B. (2013). *The C++ programming language.* (4a. ed.). Addison-Wesley.

JAVA

Deitel, P. y Deitel, H. (2017). *Java. How to program.* (11a. ed.). Pearson.
Farrell, J. (2019). *Java programming.* (9a. ed.). CENGATE.
Joyanes, L. (2014). *Programación en C/C++, Java y UML.* (2a. ed.). México.
Schildt, H. (2018). *Java. A beginner's guide. Create, compile and run Java programs.* (8a. ed.). Oracle/McGraw-Hill.
Schildt, H. (2019). *The complete reference Java. Comprehensive coverage of the Java language.* (11a. ed.). McGraw-Hill/Oracle Press.

PYTHON

Bahit, E. (2012). *Curso Python para principiantes.* Creative Commons. www.eugeniabahit.com. *Libro de problemas y talleres.*
Bayley, V. (2015). *Introduction with Python. Python programming for beginners.* Software Development Training.
Chazallet, S. (2016). *Python 3. Los fundamentos del lenguaje.* (2a. ed.). ENI.
curso-python.eugeniabahit.cm/sources.tar.gz
Deitel, P. y Deitel, H. (2019). *Intro to Python for computer science and data science: learning to program with AI, Big Data and the Cloud.* Pearson.
Deitel, P. y Deitel, H. (2019). *Python for programmers with Big Data and Artificial Intelligence case studies.* Pearson.
Esel, F., Rohaut, S. *Algoritmia: Técnicas fundamentales en programación. Ejemplos en Python.* Ediciones ENI, 2019.
Gries, P., Campbell, J. y Montojo, J. *Practical programming. An introduction to computer science using Python 3.6.* (3a. ed.). Practical Programmers version 1.0, diciembre 2017.
Halterman, R. L. (2019). *Fundamentals of Python programming.* Southern Adventist University (Draft, 9 de julio 2019). Editor: Richard L. Halterman.
Jaworski, M. y Ziadé, T. (2019). *Expert Python programming.* (3a. ed.). Packt Publishing.
Marzal, A., Gracia, I. y García, P. (2014). *Introducción a la programación con Python 3.* Universitat Jaume I.
Python. Página oficial: www.python.org
Shaw, Z. A. (2017). *Aprenda a programar en Python 3.* Anaya Multimedia.
Sheridan, Ch. (2016). Python. *Language reference manual.* Ed. Chris Sheridan.
Van Rossum (1991). Creador de Python. www.python.org/~guide. El Tutorial de Python, Traducción del Grupo de Usuarios de Python de Argentina. Octubre 2017. Tutorial.python.org.ar.
Zelle, J. (2017). *Python programming: An introduction to computer science.* (3a. ed.). Franklin, Beedle (www.fbeedle.com).

PORTALES BIBLIOGRÁFICOS/BUSCADORES ACADÉMICOS/REVISTAS ACADÉMICAS

Academia.edu	www.academia.edu
DIALNET	dialnet.unirioja.es
Google Scholar (Google académico)	scholar.google.es
Refseek	refseek.com (motor de búsqueda para estudiantes e investigadores)
Scholarpedia	scholarpedia.org

Cristian Pilaparita 1095211-2202112353

SCIELO (Scientific Electronic Library Online):	www.scielo.org
SCIELO España	http://scielo.isciii.es/scielo.php
Springer Link	link.springer.com
World Wide Science	worldwidescience.org

REPOSITORIOS INDEXADOS

Archivo Digital UPM:
oa.upm.es

CERN Document Server (Acceso a artículos, informes y contenidos multimedia):
cds.cern.ch

Depósito de Investigación Universidad de Sevilla idus:
Idus.us.es/xmlui

DIGITAL CSIC:
digital.csic.es

Microsoft Academy:
academic.microsoft.com

Portal de acceso abierto al conocimiento de la UPC:
UPCommons
Upcommons.upc.edu

Repositorio Digital CSIC:
digital.csic.es

Repositorio Institucional de la Universidad de Alicante, RUA:
rua.ua.es/dspace

Repositorio Institucional Universidad de Valladolid, UVaDoc:
uvadoc.uva.es

Repositorio Institucional Universidad Politécnica de Valencia, Riunet:
Riunet.upv.es

PLATAFORMAS DE DESARROLLO DE SOFTWARE / COMUNIDAD DE DESARROLLADORES

Plataforma de software libre / *open source* (adquirida por Microsoft):

GitHub:	www.github.com
Plataforma de Google: Chrome DevTools	developers.google.com
Plataforma Trello	www.trello.com

RECURSOS DE PROGRAMACIÓN EN INTERNET

Microsoft en educación:
www.microsoft.com/es-es/education

Oracle:
Java Oracle Academy: https://academy.oracle.com/es/solutions-java.html
oracle.com/es/java
oracle.com/es/downloads
www.java.com

Cristian Pilaparta 1095211-2202112353

ENTORNOS INTEGRADOS DE DESARROLLO (EID, IDE)

Eclipse:	www.eclipse.org
IntelliJ IEA:	www.jetbrains.com
Jetbrains:	www.jetbrains.com
NetBeans:	www.netbeans.org
Visual Studio:	https://visualstudio.microsoft.com/es/

PLATAFORMAS DE ENSEÑANZA / MOOC (MASSIVE OPEN ONLINE COURSE)

Aprender Gratis:	aprendergratis.es	
CodeAcademy:	www.codeacademy.com	
Coursera:	coursera.org	es.coursera.org
Edutech:	edutechsv.com	
EdX:	edx.org	
Future Learn:	futurelearn.com	
Khan Academy:	es.khanacademy.org	
Lynda:	www.lynda.com	
MiriadaX:	miriadax.net	
OpenHPI:	open.hpi.de	
Red Educa:	www.rededuca.net	
Tutellus:	www.tutellus.com	
Udacity:	www.udacity.com	
Udemy:	www.udemy.com	
UNED Abierta:	iedra.uned.es	

Cristian Pilapanta 1095211-2202112353

1095211220211235

C/C++

C++ Reference. Página Web con recursos disponibles en C++
www.cplusplus.com/reference
en.cppreference.com

PYTHON

- Beginner's guide to Python. https://wiki.python.org/moin/BeginnersGuide
- El tutorial de Python. Original de Guido van Rossum. http://tutorial.python.org.ar (octubre 2017). Docs.python.org.ar/tutorial/pdfs/TutorialPython3.pdf
- Eugenia Bahit. Licencia Creative Commons: www.eugeniabahit.com
  Curso de Python para principiantes
  www.iaa.csic.es/python/curso-python-para-principiantes.pdf
- *Guía de estilo del código Python.* Original de Guido van Rossum y Barry Warsaw. Traducido al español por Raúl González Duque: mundogeek.net/traducciones/guía-etilo-python.htm.
- Raúl González Duque. Python para todos
  Launchpadlibrarian.net/1898'633/Python para todos.pdf @CreativeCommons
- Sitio oficial de descarga de Python: https://www.python.org/downloads/
- Sitio oficial de documentación de Python: https://www.python.org/doc/
- Traducción de la Comunidad de Python Argentina: www.python.org.ar
- Tutorial de Python 'Python para todos':
  http://www.utic.edu.py/citil/images/Manuales/Python_para_todos.pdf
  Python para todos es un libro sobre programación en **Python** escrito por Raúl González Duque. Se trata de un **tutorial de Python** adecuado para todos los niveles y que puedes descargar totalmente gratis:
  https://launchpadlibrarian.net/18980633/Python%20para%20todos.pdf

- Victor Osetskyi (2017). *SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral*. https://medium.com/existek/sdlc-models-explained-agile-waterfall-v-shaped-iterative-spiral-e3f012f390c5

CURRÍCULO DE COMPUTACIÓN, INFORMÁTICA E INGENIERÍA DE SISTEMAS

ACM/IEEE: www.acm.org; https://dl.acm.org/doi/10.1145/2534860

Computing Curricula 2005. The Overview Report covering undergraduate degree programs in Computer Engineering, Computer Science, Information Systems, Information Technology, Software Engineering. A volume of the Computing Curricula Series the Joint Task Force for Computing Curricula 2005. A cooperative project of The Association for Computing Machinery (ACM) The Association for Information Systems (AIS) The Computer Society (IEEE-CS) 30 September 2005, https://www.cc2020.net/wp-content/uploads/user_uploads/Admin/cc2005_march06final.pdf

F. J. García-Peñalvo, "Los estudios de ingeniería en informática", en Proyecto Docente e Investigador. Catedrático de Universidad, pp. 225-275, Salamanca, España: Departamento de Informática y Automática. Universidad de Salamanca, 2018.

The Computing Curricula 2020 (CC2020) Project, What is CC2020?

Cristian Pilapaña 1095211-2202112353